Software Architecture of the TAC Energy Trading Broker

Puthyrak Kang
University of Minnesota
Minneapolis, MN

August 5, 2010

Abstract

The TAC Energy Trading Broker is a design framework on which to build working software agents. The framework will include sufficient capability to act as a broker in the TAC Energy scenario with minimal capabilities. The broker framework is designed to lower the barrier to entry that, in other TAC scenarios, imposes huge challenges for agent developers who are not strong software developers. The framework leverages the Repast Simphony toolkit so that the developers can build their brokers totally using the Repast's graphical interface. The framework also provides APIs for those developers who want to build their brokers from scratch. Additionally, the framework design is flexible, extensible, and its components will be independently testable. This makes the framework easy for framework developers to understand and maintain.

1 Introduction

The current energy market is mainly made of a grid of a small number of large plants, generators and distributors. It includes a small share of renewable energy sources. Its control structure is centralized and managed to balance the energy production and consumption. However, the existing structure leaves distributed intermittent sources of renewable power virtually uncontrollable.

The market paradigm will change. There are compelling needs for more renewal sources of energy to reduce oil and natural gas dependency. By 2020, the renewable sources will contribute to 35% of overall energy demands in the European Union [Block et al., 2009]. This will push the current market infrastructure to its limit, as it is not able to effectively manage small-scale distributed and sources of intermittent energy. To meet this challenge, it needs a control infrastructure that is more flexible, decentralized and self organizing. This structure must be able to balance both the large grid as a whole as well as the many lower voltage sub-grids including intermittent energy.

A competitive market is one candidate that can address the above challenge. This market approach offers flexibility, decentralization and self organization. It delegates responsibility to market actors to balance the energy production and consumption. However, the market-based control structure comes with risks. These risks come from the difficulty in predicting emergent phenomena. These phenomena result from interactions among market actors. Understanding and mitigating these risks needs a simulation tool that can discover emergent the phenomena and evaluate market-based-control strategies. TAC Energy [Block et al., 2009] offers a way to simulate the energy market competition using a multi-agent trading competitive simulation. TAC Energy creates a competitive testbed for researchers and domain experts to develop and assess their trading techniques and strategies.

This paper describe a design framework that creates a proto-broker with minimal capabilities to compete in the TAC Energy simulation. This framework also provides a foundation upon which researchers and domain experts can build more complex and competitive brokers.

The structure of this paper is organized as follows. In section 2, we discuss related works in the area of Agent-Based Modeling and Simulation (ABMS) design and implementations. Section 3 discusses TAC
Energy infrastructure, competition rules and broker decisions. Section 4 covers architectural design of brokers for TAC Energy. Section 5 describes broker’s behaviors in response to competition scenario in form of sequence diagrams. Section 6 covers evaluation of the design. Finally, section 7 concludes.

2 Related work

2.1 What Is Agent?

Even though there is no universal agreement on precise definition of the term “agent,” definitions seem to agree that an agent is a component (software, model, individual, etc.) that contains certain properties. Agent properties range from being able to perceive environments through sensors [Russell and Norvig, 1995] to capabilities to learn from their environment [Bonabeau, 2002]; from being controlled by primitive reactive set of behavioral rules to controlled by complex artificial intelligence (AI), and from being independent to autonomous and adaptive.

Some modelers define the term “agent” in accordance with the domain in which the agent operates. In business and social domains, an agent is defined as a component that acts on behalf of a “principal,” is uniquely identifiable, situated, goal-directed, self-directed, and flexible [Macal and North, 2005]. MinneTAC agent [Collins et al., 2008] is defined as a software component made of a set of “roles” and configuration files that map roles to the code that implements them. In TAC Energy, an agent acting as “broker” is a software component that represents a trading agent with a role as an “aggregator” of energy supply and demand [Block et al., 2009].

2.2 Agents in practices

Agents are the foundation of ABMS applications. Agent’s behaviors might be evolving and their collective behavior may not be not easily anticipated. Their interactions thus create emergent phenomena that traditional modeling techniques are not able to capture. This gives ABMS benefits over other modeling techniques in terms of (i) capturing emergent phenomena; (ii) providing a natural description of a system; and (iii) flexibility [Bonabeau, 2002].

Agents are also adopted in many commercial applications. eBay uses agents to allow customers to automate the bidding processing. Shopbot is another implementation of agent. Shopbot automatically searches for prices and product characteristics from on-line retailers [Bonabeau, 2002]. Software companies and data centers implement agents in license management system; these agents regulate agreements between subscribers and providers [Zhao et al., 2007].

2.3 Agent design

Many designers model an agent as an entity that has attributes, behaviors, set of rules and resources. Attributes make the agent unique and capable to perceive an environment. The behaviors allow the agent to interact with the environment. The rules determine how the agent makes decisions and adjusts behaviors in response to changes in the environment. The resources enable the agent to save the state of its operations.

Usually designers abstract properties of an agent into attributes, behaviors, decision making rules, learning capacities and adaptability, and resources. Not all agents have the same level of sophistication and complexity. The level of sophistication and complexity depends on what features designers decide to equip agents with. This decision is based on functional requirements of agents. MinneTAC agent supports “decision making rules;” the rules are configurable. It has limited learning capabilities, adaptability and resources.

2.4 ABMS Design challenge and barriers of entries

ABMS comes with some issues. Although common to all modeling techniques, one issue relates to the purpose of the model; a model is only as useful as the purpose for which it was constructed. Due to complexity of ABMS and agent design, the purpose of the model may be overstated. This leads to un-useful or misleading
outputs. Modelers of ABMS applications need to pay great care of what they promise [Jones, 2007]. They should resist the temptation to overstate the implication of the model outcomes, carefully design the models with an eye toward validity and provide precise specifications and results subjected to repeated tests.

Another issue is the robustness of the simulation result. The agent-based models consider the system from the perspective of its constituent unit’s activities at a disaggregated level [Bonabeau, 2002]. This level of details involves description of how many attributes and behaviors the agent may have, and how the agent interacts with its environment. This requires multiple repeated runs of simulation, systematically varying initial sets, conditions or parameters in order to achieve the robustness of the results. This process can be extremely computationally intensive and time consuming. Generated data can be also enormous, and thus complicates the analysis.

Level of sensitivity to small variations in initial conditions and interactions rules may cause the simulation result deviate from the real world.

Capturing the agent’s processing, state and interactions is another challenge in agent design. These data are important for researchers and developers to do off-line analysis to answer strategic questions. These questions include (i) how the agent processes data; (ii) what data the agent processes and uses for decision making (iii) what are intermediate results of that processing; (iv) what are the states of agent and the world; and (v) how agent responds to changes in the world. Answering these questions, agent developers can evaluate their solutions and compare which solution works better.

[Bonabeau, 2002] suggests that a model has to serve a specific purpose; a general purpose does not work. The model has to be built at a right level of descriptions; just right amount of details to meet its purpose and fit to the system it models.

To hurdle the issues, many ABMS application designers tend to develop a simulation framework or platform that coordinates and facilitates interactions among agents, and set a minimum level of the intelligence of the agents. All participants develop their own agents using the application programming interface (AIP) or specifications provided by the framework. Supply-Chain Trading Agent Competition (TAC-SCM) [Collins et al., 2010] develops a Web-based multi-agent simulation platform that captures the complexity, stochasticity and competitive nature inherent to supply chain environment. Participants, through agents, use the results from the competition simulation to assess their trading strategies. They design and implement agents independently from a variety of perspectives, individual interests and backgrounds. This creates the diversity and heterogeneity of the agents. TAC-SCM game organizers provide the agentware package that forms the foundation of each agent, and provides the ability to communicate with the game server. A common approach to construct an agent for TAC SCM is to extend classes of the agentware package directly.

Non-computer-science researchers and domain experts face significant challenges if they are forced to code agents at low level. The actual process of writing software is a complicated technical task and also prone to errors. This forms barriers to entry for researchers and domain experts. This barrier to entry potentially drives away many experts from the simulations, or forces good researchers or scientists to become bad programmers. Often, their software applications are not well designed, reusable, or easily understood [Minar et al., 1996].

2.5 Lower the barriers of entry

There are many attempts to remove or lower the barriers of entry by designers who wish to accommodate possibilities for public contributions at both framework and agent levels.

Substantial research has been done to create environments and frameworks for ABMS developments, aiming at helping non-computer people to build their agents with minimum coding efforts. Among free ABMS software development environment, Swarm [Minar et al., 1996], and Repast [North et al., 2006] are popular.

Swarm was first launched as the first ABMS software development environment in 1994 at the Sata Fe Institution. It is a multi-agent software platform for simulation of complex adaptive systems. It is intended to provide scientists a standardized suite of software tools to develop their simulations.

Repast Simphony provides a agent-development toolkit which enables simulation developers to build agents and configure their behaviors. The developers can use the toolkit’s graphical user interfaces and use the
Repast API libraries. The graphical user interface is integrated with the Eclipse IDE. The API is compatible with Java and Groovy. It has been widely used to study wholesale electricity markets [North et al., 2002, Conzelmann et al., 2004, Sun and Tesfatsion, 2007].

3 TAC Energy

3.1 TAC Energy Infrastructure

The infrastructure of the TAC Energy is made of independent components. These components include the regional market, Independent System Operator (ISO), trading broker, Market Intelligence, and customer. This section covers communications between brokers and other components. Details of the infrastructure and other components are not in the scope of this paper. They are available in TAC Energy documentation [Block et al., 2009]. However, this section provides high level descriptions of other components to help design discussions of the broker framework.

The **regional market** is the TAC Energy server that drives the competition simulation. The server works a market at which agents may buy and sell specific quantity of energy in future slots. The **Independent System Operator (ISO)** is an independent component. The ISO is responsible for ensuring exact balance between supply and demand in the real time. The **Market Intelligence** is a service component that provides brokers with information about Customers. The **Customer** is a local producer or consumer that may sell or buy energy to or from a broker.

The broker component communicates with other components through asynchronous communication. Apache ActiveMQ technology is used to support this message communication. A broker sends messages to other components through request queues and receives messages through response queues (see Figure 1). Both request and response messages are uniquely identified by their assigned identifiers. Response message’s header stores the identifier of the request to which it responds. Individual message has an expiration time. The message will be invalid if the time expires.

3.2 Competition rules

TAC Energy simulations focus on broker roles as “aggregator” of energy production and consumption which need to be established, balanced and generating profits. TAC Energy organizes the simulations through a series of alternating **contracting** and **execution** phases. In the contracting phase, brokers are required to build portfolios of energy producers and consumers. During the execution, brokers are competing to balance their energy consumption against production under contracts.
Brokers must follow competition rules as described below in creating the portfolios and balancing the production and consumption.

a. **Contract negotiation**: this is a process by which a broker enters into contracts with individual customers on energy production or consumption. Negotiation protocols are predefined by the TAC Energy. Brokers can negotiate contracts with local producers to buy energy production capacities (building portfolio of producers), and with local consumers to sell energy (building portfolio of consumers). The contract negotiation is allowed only during the contracting phase.

b. **Tariff offering**: tariff is an offered contract that may be accepted or not by anonymous customers. Tariff offer is a process by which brokers can offer tariffs to customers. The process and protocol of the offering are defined and controlled by the Market Intelligence. Brokers can use tariff offering to build their portfolios of producers and consumers. This process is allowed only in the contracting phase.

c. **Energy production tariff**: this tariff allows brokers to make agreements with distributed energy generators such as Combine Head and Power (CHP) plant to buy energy. The generators are paid for operating the energy generation and feeding it to the grid. This happens during the contracting phase only.

d. **PEV tariff**: this is a special type of tariffs that brokers can sell to Plug-in Electric Vehicle (PEVs). Under these contracts, prices may be time-dependent.

e. **Loading control influence**: brokers do not have direct control of loading capacities of customers under contract. However, brokers can influence the capacities of their own producers and consumers through pricing policies. For instance, if brokers want to encourage their production, they can increase production prices for their producers with flexible contracts. And, if they want to discourage consumption, they can increase consumption prices. Brokers are allowed to use this policy during the execution phase.

f. **Trading**: During the execution, brokers can trade their energy in the Regional Market. The process and protocol are determined and controlled by the Regional Market.

From a broker’s perspective, the ultimate goal in the competition is (i) to negotiate tariffs and contracts to create quality portfolios during the contracting phase; and (ii) to balance them during the execution phase while optimizing its financial performance. The quality of a portfolio is driven by the profitability in expectation and the flexibility that the broker has to easily balance the portfolio during the execution.

### 3.3 Agent decision

In the contracting phase, broker’s decision includes (i) learning about customer preferences; (ii) forecasting future production and consumption; (iii) acquiring capacity; and (iv) selling energy. During the execution phase, the decision includes balancing its consumption against production in contract.

During the contracting phase, the broker learns preferences of its customers from information fed by the Market Intelligence. This information includes future time series of intermittent generators, and historical time series for other customers. With this knowledge, the broker predicts the future production and consumption of its portfolios. The broker acquires production capacities by negotiating tariffs and contracts with local producers. As a result, the broker creates a producer portfolio. The broker can sell energy to local consumers by negotiating consumption contracts and offering tariffs. This creates a consumer portfolio for the broker.

During the execution phase, a broker manages its portfolio to balance its consumption against production under contract. The broker can resolve any imbalance by trading in the Regional Market or by adjusting energy prices for its customers that have flexible contracts.
4 Architectural design of TAC Energy broker framework

4.1 Audiences

The broker framework targets three main audiences. These groups include *domain-oriented agent developers*, *software-oriented agent developers* and *framework developers*. The *domain-oriented agent developers* include non-computer-science researchers and domain experts. These people are interested in developing and evaluating their trading techniques and strategies. They need a toolkit that enables to build competitive brokers for the TAC Energy simulations with minimum coding.

The *software-oriented agent developers* are researchers with computer-science background. List domain-oriented agent developers, the software-oriented developers are also interested in developing and evaluating their trading techniques and strategies. But, they prefer to use the framework APIs to build their broker agents. They need the API library that are flexible, extensible and easy to understand. The library should also provide testability.

Both groups of agent developers also require that the framework provide them sufficient log information to perform off-line analysis of the broker’s behaviors, decision making and states of environments. With analysis, they can assess and compare their trading techniques and strategies.

The *framework developers* include programmers who implement this framework architecture, and other computer-science researchers who are interested in contributing to this framework design and implementation. The programmers implementing the frameworks require that the architecture be implementable for a reason amount of resources and time. Other researchers who want to contribute to the framework require that the framework be easy to understand, modify, maintain and test. The framework shall provide sufficient execution log for them to debug any run-time error or performance issues.

4.2 Design objective, quality attributes and design principles

The broker framework design aims at addressing the needs of each audience group discussed in section 4.1. To reach this goal, the design shall lower the barriers of entries discussed in section 2.4. In other word, the design shall create a foundational framework upon which agent developers can build their brokers using a graphical configuration tool or the framework API. The design shall also make the framework easy to understand, modify, maintain and test while preserving the design conceptual integrity. To facilitate the framework learning and understanding, the framework shall provide a working a proto-broker which has minimum capability to compete in TAC Energy competitive market. Agent developers can deploy the proto-broker as it to the simulation to understand how the broker interacts with other components. They can also use the proto-broker as a sample broker helping them build their brokers.

In addition, the framework shall provide sufficient data and execution logs for off-line analysis and solution evaluations.

To meet the above goal, the framework design shall be driven by two main quality attributes: *flexibility* and *extensibility*. These attributes enable the broker design to adapt to uncertainty and complexity of market competitions.

The framework shall comply with a number of main design principles. These principles include *(i) asynchronous message communication, (ii) total exposure to the Repast S, (iii) consistency in error handling, and (iv) research-oriented logger.*

4.3 Conceptual model of the framework

Figure 2 depicts a high-level conceptual view of the framework design. The figure shows two barriers of entries. One barrier is for domain-oriented agent developers. This barrier is represented by the horizontal line separating the domain experts and the TAC Engine infrastructure. The other barrier is for software-oriented agent developers. This barrier is presented by the vertical line that separates framework developers from the TAC Energy infrastructure.
To lower this barrier for domain-oriented developers, the framework provides the Configuration Interface that can work with a third-party configuration tool (for example, Repast Simphony toolkit). This third-party tool provides graphical user interface for agent developers to build a broker and script its behavior. The tool saves the developer works into a configuration data. This configuration data includes data and execution codes. The Configuration Interface will translate and convert the third-party configuration data into the framework configuration data structure. The framework uses this data structure to instantiate the broker’s attributes, behaviors and decision making rules.

To lower the barrier for software-oriented developers, the framework provides the API packages that contain interfaces and implementations necessary to build and configure a complex broker.

### 4.4 Component design

The term “Component” is often used interchangeably with “Object,” “Module” in software design. To help our discussion, it is important to clarify this term.

The “component” is a unit of software system that has the following characteristic properties: (i) independently deployed; (ii) composable with third-party components; and (iii) having no persistent state [Szyperski, 1998]. For a component to be independently deployed, it must be well separated from its environments and other components. It must encapsulate its features, data structures and construction details, and never be deployed partially. For a component to be composable with other third-party components, it needs to be sufficiently self-contained. It also needs to come with a clear specification of what it requires and provides, and how it interacts with environments. For a component not have any persistent state, it requires that the component cannot be distinguished from copies of its own. A single component can be loaded into and activated in a particular system.

The “module” is known as a package in the object-oriented design. Each module or package contains multiple classes. Each class can inherit from other class across package boundaries.

Figure 3 shows the component design of the broker framework and its interactions with other TAC Energy components. A gray box represents a component in the TAC Energy infrastructure. A light blue box is a sub-component of the broker component. A white box represents a module. A solid arrow represents an internal
interface between broker components. A dash arrow encapsulates asynchronous message communication between TAC Energy components. This section focuses on the component design of the broker.

This design breaks down the broker component from three main aspects. These aspects include (i) processing; (ii) state; and (iii) interaction. The processing encapsulates how the broker processes data and makes decisions. This corresponds to the main-brain agent property. The state is the data processed by the broker and a snapshot of the world at a particular time. This is equivalent to the data-source property of an agent. The interaction is a communication model between the broker and an individual component in the TAC Energy framework. Each aspect is implemented as a sub-component or module. Sub-components include Portfolio Manager, Trader, Market Message Listener, Supply and Demand Predictor, Tariff Negotiator, and Contract Negotiator. Modules include Knowledge Base, Configuration Interface, Connection Manager, Error Handler, and Logger.

Portfolio Manager is the main subcomponent of the broker agent. It embraces intelligence that the broker needs to perform actions. It is responsible for establishing the quality portfolios (see section 3.2) during the contract phase, and managing the portfolios to balance production and consumption during the execution. The Portfolio Manager contains two modules: Risk Manager and Learning Capability. The Risk manager is responsible for quantifying and mitigating both load risks and financial risks while the broker applies its balancing policies. The Risk Manager can also hedge the portfolios against future contracts in the Regional Market. The Learning Capability provides ability for the broker to gather knowledges of client preferences through the market intelligence, and knowledge of trading through market notifications. The broker uses this knowledge to update states and strategies in the Knowledge Base.

The Knowledge Base is a repository that stores data of rules and facts which can static or dynamic. The rules are logical conditions that determine decision making by the broker. The facts are the states of the broker and the world. The static data are generated by the agent-development toolkit. The dynamic data are captured during the simulation runtime.

The Supply and Demand Predictor is responsible for gathering information about the customer preferences and predicting future energy demands and supplies. Customer preferences are received from the
Market Intelligent.

The Contract Negotiator encapsulates communications between the broker and the contract customers. The communication is asynchronous and uses a pre-defined protocol. Through the Contract Negotiator, the broker negotiates contracts production and consumption contracts with customers.

Tariff Negotiator provides asynchronous communication between the broker and the Market Intelligence. The communication protocol is defined and controlled by the Market Intelligence. The Tariff Negotiator enables the broker to offer tariffs to customers through the Market Intelligence.

The Trader is responsible for bidding or selling energy in the regional market. It is usually done during the execution phase. It can also take place during the contracting phase when the broker decides to hedge its portfolio against the future contracts in the regional market.

Market Message Listener is a component that listens to notifications sent by the Regional Market.

There are a number of infrastructure components of the broker framework. These components include Configuration Interface, Connection Manager, Error Handler and Logger.

The Configuration Interface provides specification of the framework configuration data structure. This data structure is used to build the broker’s attributes, behaviors, knowledge base and decision making rules. A concrete implementation of this interface can act as an adaptor between the broker framework and a third-party toolkit (e.g. the Repast). The concrete implementation will convert the configuration data generated by the third-part tool to the framework configuration data structure.

The Connection Manager is responsible for creating and maintaining connection objects to the knowledge-base repositories, and all connections to Apache ActiveMQs. Error Handler is a component that handles the error states of the brokers as discussed in section 4.9. Logger is responsible for logging information into specified log destinations, as discussed in section 4.8.

### 4.5 Package structures

![Broker package framework](image)

Figure 4: Broker package framework

Figure 4 displays a package structure of the broker framework which is discussed at high-level in section 4.3. The package structure breaks down the components in section 4.4 in packages and groups them into **Core Packages, Internal Packages, Plug-in Packages, and Infrastructure Packages**.
The core packages encapsulates the design and implementation of the framework’s backbone. At this level, it is not a design for change, even though their internal components are not tightly coupled among themselves. Any changes to these packages have high potential impacts on the framework structure.

The framework’s backbone is the skeleton of the broker connecting the sensors, the central brain, and the effectors. The sensors are made of the market message listeners and supply demand predictors. The central brain contains portfolio manager and risk manager. The effectors include tariff negotiator, contract negotiator and trader.

The Internal packages contain the implementations of the sensors (predicting algorithm), the configuration setup (broker configuration), and the implementation of the brain (risk management, knowledge base and learning capacity). Changes in these packages does not modify the framework skeleton, but only the way the broker perceives the world.

The API packages provide interfaces and implementations necessary to build a complex and competitive broker.

The Infrastructure packages provide implementations of Connection Manager, Market Listener, Logger and Error Handlers. These packages also include the specification of the framework configuration interface.

The Repast plug-in is an implementation of the framework configuration interface. It is an adapter which makes the framework works seamlessly with the Repast Simphony.

4.6 Events Model

Event-based architecture is a design style in which software components communicate with each other via explicit software connectors using events, or messages [Hendrickson et al., 2005]. Events are discrete data objects that do not contain direct pointers to data in memory or control entities like thread objects. Usually, event-based systems are asynchronous—a component may send or receive events at any time. Such system has many beneficial characteristics such as low coupling, ease of dynamic configuration and ease of distribution across multiple heterogeneous platforms.

TAC Energy broker is both reactive and initiative system. It is reactive in the sense that it responds to events sent from other TAC Energy components. It is initiative in the sense that it generates events for other components. The events are in the form of asynchronous messages that the broker receives from or sends to other components at any time.

4.6.1 Reactive mode

The broker is reactive to the following events—game-starts event, RFQ event, tariff-offer-request event, currently-offered-tariffs event, time-slot events, notification event, end-of-execution event, and game-ends event, as shown in Figure 5.

The game-starts event is sent by the TAC Energy server. This event notifies brokers that the game now enters the contracting phase. In response, brokers change their states to contracting states, and ready to build their portfolios.

The RFQ events are sent from local producers during the contracting phase. These events notify the broker that there are Request for Quotes (RFQ) available in the local producer market. In response to these events, the broker estimates the future demands and supply and decides if it wants to make a deal (as show in section 5.3).

The tariff-offer-request events are received from the Market Intelligence during the contracting phase. These events notify the broker that there are Request for Quotes (RFQ) available in the local producer market. In response to these events, the broker engages in the tariff offering process if it decides to (see section 5.4).

The currently-offered-tariff events are broadcasted by the Market Intelligence after a tariff-offer is settled. These events happen during the contracting phase. The broker responds to these events by updating its knowledge base and strategies to negotiate a tariff offer.

The time-slot events are triggered when a time slot starts and ends during the execution phase. For the first time-slot event, each broker needs to move from contracting state to execution state. Then, the broker
estimates future supplies and demands, and evaluates current status of its portfolios (section 5.2). If the portfolios are imbalance, the broker needs to take proper actions to adjust them (section 5.5, 5.6, 5.7).

The notification events are triggered when the market broadcasts limit orders or market orders to all brokers. When receiving a message, brokers analyze it and take actions in response (section 5.8).

The end-of-execution events are triggered when the TAC Energy notifies all agents that the execution phase is done. The broker moves back to the contracting state, and starts rebuilding its portfolios.

The game-ends event is sent by the TAC Energy server to notify brokers that the game is over.

4.6.2 Initiative mode

The broker generates events during requesting time series from the Market Intelligent, negotiating tariffs and contracts, and trading in the Regional Market.

The broker sends request messages to the Market Intelligence for future or historical time series for its customers (see section 5.2).

During contract negotiation, the broker sends two types of events: offer and status (see section 5.3). An offer event informs contract customers that the broker is willing to accept the negotiated RFQ at a certain price and under certain conditions. The status event informs the other negotiating party about the final decision of the broker. The final decision can be ACCEPT, REJECT or WITHDRAW.

During tariff negotiation, the broker triggers two events: tariff-offer and status (see section 5.4). The tariff-offer events notify the Market Intelligence that the broker submits a list of tariff offers. The status events notify the Market Intelligence that the broker decides to either ACCEPT or WITHDRAW the offers desired by tariff customers.
During trading in the execution simulation phase, the broker can initiate a shout events as discussed in section 5.7. A shout event can be either bid or ask. The bid is an offer to buy a market order or limit order. The ask is an offer to sell a market order or limit order.

4.7 Repast Plugin

Repast plug-in is an implementation of the Configuration Interface. The plug-in allows the framework to read configuration setup generated by Repast Simphony (Repast S). An agent developer uses Repast’s Toolkit to create agent properties and behaviors. The framework extracts information provided by Repast and creates a broker agent for the developer. From the developer’s perspective, the framework is seamlessly integrated with the Repast toolkit.

Repast Simphony (Repast S) provides an development environment for an agent-based simulation. A model created by Repast toolkit has a standard structure. This structure is based on two core data structures which include Context and Projections. The Context is a simple container of agents. The Context does not provide any mechanism for interactions between its member. These interactions are defined and enforced by the Projections. Each project is designed to work with an arbitrary object. As a result, switching projections is simple. No code changes are required to allow a projection work with a particular agent.

In the broker design framework, the Repast plug-in maps a Context to the portfolio manager, projections to the broker’s central brain and agents to the broker’s effectors.

In Repast, Context has hierarchical structure. A Context may contain sub-contexts. This hierarchy does not exist in the broker framework.

4.8 Logger

The framework’s logger provides brokers with logging capability. The logger separates logging messages into two main categories: data and execution. The data is information about operation states of each brokers. The execution log records traces of broker operations.

The data is used to analyze and evaluate effectiveness of the broker’s sensors, learning capabilities and effectors. Modelers may be interested in evaluating how effective the broker predicts future demands and supplies. Modelers may also want to assess their trading strategies that guide the broker to do its trading.

The execution log enables modelers to analyze any exception encountered by the broker during simulation. Modelers can use the execution log to improve run-time performance of the broker software.

4.9 Error Handler

A broker moves from a healthy state to an error state when execution of broker code encounter an error. The healthy state includes a normal path of the broker that starts from the initialization state through alternative contracting state and the execution state to the end state (see Figure 6). The broker stays in the execution state until the simulation finishes. Then, the broker moves to the final state. The error state includes impairment, outage, inactive, and fatal termination.

When an error occurs and the current state is initialization, the broker is moved to fatal state. Otherwise, the broker is moved to impairment state. During this state transition, the framework error handler is called to handle the error condition.

If the current state of the broker is fatal termination, the error handler releases all resources currently occupied by the broker, and kills the broker component. This scenario happens when the broker fails to instantiate all required resources to setup its attributes and behaviors. The required resources include configuration data, connection to the knowledge-base repository and MQ connections to other components in the TAC Energy infrastructure.

If the current state of the broker is impairment, the error handler alerts the broker’s supports. If the error relates to refreshing the resource, the handler changes the broker state to “outage.” In the impairment state, the broker is still able to communicate with other TAC Energy components, but its behavior is not predictable. The handler keeps notifying supports and refreshing the error status.
In the outage state, the broker is temporarily unable to participate the simulation. The handler tries to reload the failing resources. If it still fails after a certain number of tries, the handler changes the broker state to “inactive.”

In inactive state, the broker is presumably terminated, even though its instance is still available in the TAC Energy’s virtual machine. The handler will not try to do any recovery. From simulation perspective, the broker is out from the game. The broker component will be terminated along with the virtual machine of the simulation. The handler will release all resources that the broker currently holds.

5 TAC Energy Broker: Dynamic Model

In the following sequence diagram, the gray background that covers multiple object lifelines indicates that the objects belong to the same component. For example, in Figure 10, the risk manager is an internal module of the portfolio manager, and thus it is shown inside the gray background with the portfolio manager.

A lifeline with gray header represents an external component of the TAC Energy. It encapsulates a communication between the broker and that component. For instance, in Figure 8, the “Market Intelligence” captures the communication between the broker’s Supply and Demand Predictor and the Market Intelligence. This communication is done through asynchronous message queues. The predictor sends request to the Market Intelligence through a request queue and receives response from the Market Intelligence through a response queue.

5.1 Broker software instantiates its components during the load time

Figure 7 shows how the broker software instantiates its components. The sequence diagram starts when the main method of the Broker driver class is invoked (step 1). The main method then calls the Broker’s initialization method to start the instantiation process (step 2). The Broker loads configuration data from all configuration files from locations specified in the arguments of the main method. If not specified, the broker uses the current location. The broker software terminates with error if it fails to load configurations.
The configuration data are passed along to instances of each component during the constructor calls. Each constructor extracts values from the configuration data and instantiates the constructed instance’s attributes.

In step 3, the Broker creates an instance of ConnectionManager class, passing configuration data. In step 4, the Broker creates an instance of PortfolioManager. The PortfolioManager constructor invokes its createPortfolio method to instantiate a portfolio object. In step 5 to 11, the Broker creates instances of TariffNegotiator, ContractNegotiator, SupplyDemandPredictor, Trader, Market Message Listener, Logger and Error Handler respectively.

5.2 Broker collects information and predicts future production and consumption of its customers under contracts

Figure 8 shows how the broker collects information about preferences of its customers under contracts, and predicts their future energy production and consumption.

The sequence starts when the SupplyDemandPredictor is invoked to predict production capacities and energy consumption of producers and consumers under current contracts (step 1). The predictor gets list of all producers and consumers from the portfolio manager (step 2.1 and 2.2). The predictor requests for future
Figure 8: Predicting future production and consumers of the broker’s portfolio

time series for intermittent generators (step 3.1.1, 3.1.2). The predictor requests the Market Intelligence for historical time series for other customers (step 3.2.1, 3.2.2).

5.3 Broker acquires energy production capacity from local producers

Figure 9 shows how a broker acquires energy generation capacities from local producers during the contracting phase which has a short period of time (perhaps 60-120 seconds).

The sequence starts the Contract Negotiator receives a RFQ event from the local producer (step 1). The negotiator forwards the event to the portfolio by calling processRFQs method (step 2). The portfolio manager processes the message and invokes its knowledge base to update and store information (step 3). In step 4, the portfolio manager needs to estimate and reason about consumers and producers in order to design appropriate tariffs and respond to the RFQs.

If the portfolio manager decides to negotiate with producers for the RFQs, it enters into the negotiation loop in step 5; otherwise it proceeds to step 6. In step 5.x, the portfolio manager creates an offer and invokes offer method of the contract negotiator to send offer to the local producer. The negotiator sends the offer to the producer, and waits for response from the producer through its response queue. The producer can accept the offer, submit counteroffer or withdraw the RFQs. After receiving the response from the produce, the negotiator invokes setResponseStatus method of the portfolio management to set the status of the current offer: the status can be ACCEPT, COUNTEROFFER or WITHDRAW. Now it is back to the start of the negotiation loop.

In step 6, the portfolio manager makes decision to accept, reject or withdraw the RFQs, by invoking the setRFQStatus method of the negotiator. Based on the status, the negotiator can send Accept, Reject or Withdraw message to the producer throw its request queue (step 7.x, 8).

Finally, the portfolio manager updates its portfolio objects if there is any change (step 9).
5.4 Broker offers tariff to tariff customers

Figure 10 depicts a tariff-offering process which simulates the real-world competition in which firms are bidding each other to attract most “desirable” customers to their offering. Brokers are allowed to offer tariffs in multiple “rounds,” with the number of rounds R indeterminate.

The sequence is triggered when the tariff negotiator receives a broadcasted message from the Market Intelligence. The message asks for tariff submission from each broker (step 1). The tariff negotiator notifies the portfolio manager by invoking setOpenTariffOffer method (step 1.1).

The portfolio evaluates if it is ready to submit a tariff offer. If it is ready, the portfolio manager creates tariffs (step 2) and invokes offerTariffs method of the tariff negotiator (step 3) to submit the tariffs to the market intelligence. The negotiator then sends the list of tariffs to the market intelligence (step 3.1) through its request queue and receives a response (step 3.2) from the market intelligence through its response queue. The response contains profiles of customers that desires to accept the offer, a new-tariff-offer flag and tariff-offering fee. The new-tariff-offer flag indicates whether the broker can proceed to another round of tariff-offer submission. The tariff-offering fee is accumulative fee that charges the broker for tariff offering up to now.
The negotiator forwards the customer profiles to the portfolio manager (step 4) and sets the allow-new-tariff-offer flag (step 5). If the broker is not allowed to submit another offer, or the portfolio manager decides to end the offering, the portfolio manager invokes setTariffOfferStatus method of the TariffNegotiator to send its final decision to the Market Intelligence (step 6). The negotiator sends the status to the Market Intelligence (step 7). The portfolio manager then invokes the Risk Manager to store the accumulative offering fee (step 8), and updates its portfolio objects (step 9).

5.5 Broker adjusts energy portfolio

During the execution phase, the broker is responsible to level the energy consumption of its consumers against its production capacities. It predicts future production and consumption of customers under contracts, and evaluates whether it needs to take any action based on the knowledge base. This is illustrated in Figure 11. In step 1, the portfolio manager requests the supply and demand predictor to forecast the future produc-
tion and consumption of its customers (as described in Figure 8. It then creates a percept of the environment, and calls getAction method of the knowledge base object passing the concept. The knowledge base instantiates an instance of Action which is appropriate to the information in the concept (step 3), and returns to the portfolio manager. In this sequence diagram, the appropriate action is the Internal Balance Action. The portfolio manager invokes execute method of the action object returned by the knowledge base (step 4).

5.6 Broker adjusts energy portfolio using internal capacity balancing

Figure 12 shows how the internal balancing action is executed. It also show how the balancing action adjusts the consumption price in an attempt to change the production and consumption behaviors of the customers.

In step 1, the portfolio manager invokes execute method of the internal balancing action object. The balancing object simply forwards the method call to the risk manager by invoking its adjustConsumptionPrice method (step 2). The risk manager calculates consumption price. It then calls newConsumptionPrice method of the contract negotiator to communicate the new consumption price to the contract customers (step 3). The contract negotiator invokes getAffectedCustomers method of the portfolio manager to get a list of all
affected customers (step 4). The negotiator then sends a message to the customers through the request queue notifying the new consumption prices (step 5). Finally, the risk manager invokes the portfolio manager to update its portfolios (step 6).

5.7 Broker adjusts portfolio by trading in the regional market

Figure 13: Broker adjusts its portfolio by trading in the regional market

Figure 13 shows another way to balance the portfolios. In this approach, the broker can buy missing or sell excess capacities on the regional market. In step 1, the portfolio manager invokes execute method of the energy trading action object. The action object then calculates the amount of energy missing or excessing. The action invokes tradeWithExistingOrdersFromMarket method of the knowledge base to find a matched order that currently exists in the market (step 2). If it founds one, the knowledge base returns a reference to the accepted order; otherwise returns null.

If there is no matched order found in the market, the energy tradition action invokes the trade method of the trader to submit an offer to the regional market (step 3). The trader sends a shout message to the regional market through its request queue (step 4) and receives through response queue an onOrderStatusUpdateReceived message from the regional market.

The regional market tries to find a match to the order submitted by the broker’s trader. If found, it broadcasts onQuoteUpdateReceived and onOrderBookUpdatedReceived to all brokers. The broker processes these message as discuss in section 5.8.

In step 5, the energy trading action invokes the portfolio manager to update its portfolio objects.

5.8 Broker processes broadcast message from the regional market

Figure 14 depicts how the broker processes a message broadcasted by the regional market. The sequence diagram starts when the regional market sends a notification message to the broadcasting channel (step 1). The market message listener receives the message and invokes the knowledge base to process the message (step 2). The knowledge updates its repository to store current state of the regional market. It then evaluates if any of the market notification matches any of the broker’s outstanding orders submitted by the trader in
Figure 14: Broker processes broadcast message from the regional market

section 5.7. If there is, the knowledge base invokes the energy trading action to submit the accepted order (step 3).

6 Design evaluation

Success of the broker framework is evaluated based on a number of aspects. These aspects include problem solving, usability, extensibility and some design principles.

6.1 Problem solving

Any problem solving tool needs to provide solution to the problem it tries to solve. The problem that the broker framework tries to solve is developing a fundamental framework upon which agent developers can create sophisticated broker agents. The framework also needs to provide the developers with a working proto-broker with minimum competition capabilities. The proto-broker can serve as a sample broker helping the framework learning and broker creation.

The broker framework is not a novel work. The framework leverages the agent design in the field of artificial intelligence [Russell and Norvig, 1995] to design the broker’s core packages. The framework also adopts the Repast Simphony toolkit to provide agent developers a visual configuration tool to build a broker. The component design of the framework adapts the design principles and design concepts of component-based design models, event-driven models and plug-in models. These features make the framework implementation and developing a working broker do-able and due-able. Java and Groovy are target technologies. Those programmers with experiences in these languages and object-oriented design should find the framework design easy to understand and implement.

6.2 Usability

The usability of the broker framework is defined as the ability that agent developers have to build their brokers using the framework toolkit. This is the barrier of entries that the framework attempts to lower.

Domain-oriented agent developers can use the Repast Simphony graphical toolkit to build a broker. The framework translates the configuration data generated by the Repast into a working broker. The framework captures states of the broker and states of the market. These states can be queried through the Repast interfaces.

For software-oriented agent developers, the framework offers a complete APIs. These APIs allows the agent developers to develop their brokers from low-level coding.
6.3 Extensibility

Since the framework is an open-source project, framework developers should freely access to its design and implementation. The framework should be easy to understand and easy to modify without losing the framework integrity.

The framework design satisfies this evaluation criteria. The flexibility and extensibility are the two main quality attributes that drive the framework design. Features of component-based design model and plug-in model enables programmers to easily modify or add new components to the framework. They also make the framework easy to maintain.

6.4 Other design principles

A software design must comply with certain design principles. These principles create and enforce constraints on design decisions. The principles may also determine the environment in which the software is deployed.

The broker framework design has two main design principles. These principles include (i) asynchronous message communication, (ii) total exposure to the Repast S, (iii) consistency in error handling, and (iv) research-oriented logger.

The message queue is imposed on the communication protocol between the broker and other TAC Energy components. The message queue is an asynchronous queue. The Repast integration is imposed on the framework’s configuration tool. The framework must be exposed to the Repast Simphony. Agent developers can totally build their brokers using the Repast toolkit. The consistency in error handling requires that the framework consistently handle an error and prevent the broker from crashing. The research-oriented logger is imposed on how the framework captures the broker’s state and execution data. The log data shall be sufficient for off-line analysis and evaluation.

The broker framework satisfies these design principle. The broker uses the Apache ActiveMQ for its message communication. To support asynchronous messaging, the broker sets up a pair of message queues for each communication with a component: a request queue and a response queue. For instance in Figure 1, there are a request queue and response queue to transfer messages between the broker and the contract customers. The broker sending messages through the request queue, and receives messages through the response queue. The received messages can either messages responding to its request messages or broadcasted messages. Each message has a unique identifier and a optional correlation identifier that defines the message it responds to.

The broker framework is exposed to the Repast Simphony through the Repast plug-in. The plug-in is an implementation of the framework configuration interface. This interfaces translate the data and compiled code generated by the Repast to the framework data structure that is used to build the broker attributes, behaviors and decision rules.

To satisfy the consistent error handling principle, the framework creates a module (error handler) specializing in logging. This handler puts the broker in proper state when error happens and is resolved. The handler prevents the broker from crashing. In case of un-resolved fatal error, the handler puts the broker in the inactive mode.

The framework satisfies the research-oriented log principle by creating a specialized logger. The logger logs both data and execution information. These information are available for off-line analysis and evaluation. The log format and logging process are configurable. This enables developers to customize their logging.

6.5 Testability

A design of a complex component is considered successful if the component can be tested independently from other components.

The framework design is able to provide the testability. The design enforces dependency control between the broker component and other TAC Energy components. The broker component is loosely coupled with others components due to its asynchronous message queues. This makes the broker easy to test independently.
Framework developers can easily mock any other TAC Energy components. The mocked components enable developers to test full functionality of the broker.

The framework also supports testability for the broker component and subcomponents. The broker component can be tested with its fully implemented modules and mocked subcomponents. A subcomponent can be fully tested with the mocks of other subcomponents. This is made possible because each subcomponent is loosely coupled with other sub-components. There is no direct dependency among sub-components. Subcomponent communication is done through data object interface.

7 Conclusion and future works

The Agent-Based Modeling and Simulation (ABMS) method is gaining popularity among simulation applications in wide range of fields. One benefit that ABMS has over other simulation technique is the ability to generate emergent phenomena. This benefit comes with the complexity of ABMS design and implementation. This complexity creates barriers of entries for domain experts who have limited background in computer sciences. These barriers potentially drive them away from the simulation.

This paper shows a design solution that lowers the barriers of entries in the TAC Energy domain. The design provides a broker framework that is flexible and extensible. Agent developers can build brokers using the Repast Simphony toolkit or APIs provided by the framework.

Framework developers can freely contribute to the framework design and implementation. To facilitate potential contribution, the design focuses on dependency control and modularization. It adapts the design concept and principles of component-based, event-base and plug-in design models.

The framework complies with a number of qualities attributes and design principles. These design attributes include flexibility, extensibility and testability. The design principles consist of message-queue communication and full exposure to Repast Simphony. The message-queue communication is an asynchronous message queue. The framework uses Apache ActiveMQ to support it. The full exposure to the Repast is imposed on the framework configuration tool. The Repast plug-in is an implementation of the configuration tool. This plug-in translates configuration data generated by the Repast into the configuration data structure of the framework.

The future work is implementing this framework design. The first step in the implementation is assessing technologies that this design depends on. These technologies include Apache ActiveMQ, Repast Simphony, and knowledge-based repository (Relational Database). The next step is flushing out the details of protocols of communications and negotiations. The next step will be coding. The coding will start with defining the framework APIs needed to build a broker. And then it proceeds to defining packages (plug-in, internal, core and infrastructure packages). Finally the concrete implementation starts.

8 Acknowledgment

The author would like to thank architects of TAC Energy who provide a clear specification for broker agents that will participate in the competition simulation. Requirements that drives the design discussed in this paper derives from this specification.

References


