POWERMATCHER
smartgrid technology

PowerMatcher
Agent Design and Implementation Guide

Release 0.9

Revision 1

Contents

1 Introduction

1.1 Purpose of this document

Our energy supply and infrastructure is evolving. The classic top-down approach, in which a few large power plants produce electricity that is distributed to many end-customers, is disappearing. Instead, more and more smaller, decentralized, production units are introduced, also at the end-customer's premises. Many of these decentralized generation units are intermittent, i.e. their output power is characterized by high fluctuations and sometimes difficult to predict. To maintain the stable and reliable supply of electricity, these decentralized units need to be actively coordinated. Moreover, the current system in which supply follows the demand must change, i.e. demand has to follow the supply. Thus, the demand side also needs to be actively coordinated. To make such a transition happen, a decentralized coordination mechanism is required to match supply and demand of these (smaller) units and provide interaction with existing and new markets, such as the day-ahead, intraday and balancing markets.

The PowerMatcher is a technology, that provides a decentralized coordination mechanism through multi-agent systems and electronic markets. This technology has been applied with great success in a many number of field demonstrations and simulations at various levels of the electricity provisioning. One of the key components in the PowerMatcher is the device agent, as it provides the primary interface between devices, it's stakeholders and markets.

Designing and implementing agents can be a difficult task. It requires knowledge of the physical process of a device, the boundaries set by the stakeholder (end-user) in which the device must operate and the optimization and risks of market trading strategies that can be used. Therefore, this guide has been written to help developers to take the first steps in designing and implementing PowerMatcher agents. It provides a step by step approach and includes code and mathematical examples.

1.2 Scope of this document

This guide is not exhaustive and was not meant to be. It merely outlines the first steps towards professional agent development.

This guide provides full programming code for an agent, which can be copied and pasted into a developers environment. The code examples are written in Java and use the Java PowerMatcher core version 3.1, written by IBM. This version is a full mirror of TNO's C#/.NET version 3.0.4.

It is assumed that the reader is familiar with the Java programming language and has basic mathematical knowledge.

1.3 Terms and abbreviations

Term	Description
BRP	Balance Responsible Party
CHP	Combined Heat and Power
DER	Distributed Energy Resources
DG	Distributed Generation
DR	Demand Response
DSO	Distribution System Operator
ICT	Information and Communication Technologies
kWe	kilowatt electric
kWth	kilowatt thermal
PV	Photo Voltaic
RES	Renewable Energy Sources
TSO	Transmission System Operator
VPP	Virtual Power Plant

1.4 Overview

A brief overview of the PowerMatcher concept and its components are given in chapter 2. In chapter 3, the principles of device agents are discussed, explaining the concept of device flexibility and the thought on how to use this flexibility in a market of which little is known. TNO has worked a lot with (micro-)CHP devices in field demonstrations and has created a good understanding how flexibility of these devices can be used in an optimal way. Chapter 4 therefore provides a complete description of the trade and bidding strategies for CHPs in general. An example implementation of a “dumb” agent (i.e. there is no real intelligence in the agent, nor does it represent a real device), including Java code, is described in chapter 6

Last, a list of articles and books is provided in chapter 6, to provide the reader with more background information on the PowerMatcher concept, the architectural design in field demonstrations and the results obtained.

2 The PowerMatcher

2.1 Introduction

An ICT based coordination system to cluster and control distributed energy resources (DER) must fulfil a number of requirements. The concept aims to involve every DER component actively in the cluster, up to the devices that can be found in a common household. Hence, clusters can contain several millions of DER devices, which requires a highly scalable coordination system. Herein, centralized control reaches its complexity limits. Next, the system needs to be open. DER devices should be able to connect and disconnect at will and be able to switch to other clusters. Furthermore, the energy grid is characterized by many different actors, some of them which have conflicting interests. Therefore, the coordination system must exceed the boundaries of ownership and optimize at both global and local level. It should also decide locally on local issues. Last, the system must be aligned with the liberalization of the energy markets, to allow a seamless evolution of the energy grid. Two main concepts have been identified that can fulfil these requirements: **multi-agent systems** and **electronic markets**.

2.2 Multi-agent Systems

The advanced technology of multi-agent systems (MAS) provides a well-researched way of implementing complex distributed, scalable, and open ICT systems. A multi-agent system is a system of multiple interacting software agents. A software agent is a self-contained software program that acts as a representative of something or someone (e.g., a device or a user). A software agent is goal-oriented: it carries out a task, and embodies knowledge for this purpose. For this task, it uses information from and performs actions in its local environment or context. Further, it is able to communicate with other entities (agents, systems, humans) for its tasks.

In multi-agent systems, a large number of actors are able to interact. Local agents focus on the interests of local sub-systems and influence the whole system via negotiations with other software agents. While the complexity of individual agents remains low, the intelligence level of the global system is high. In this way, multi-agent systems implement distributed decision-making systems in an open, flexible, and extensible way. Communication between actors can be minimized to a generic and uniform information exchange.

2.3 Electronic Markets

The interactions of individual agents in multi-agent systems can be made more efficient by using *electronic markets*, which provide a framework for distributed decision making based on microeconomics. Microeconomics is a branch of economics that studies how economic agents (i.e. individuals, households, and firms) make decisions to allocate limited resources, typically in markets where goods or services are being bought and sold. One of the goals of microeconomics is to analyze market mechanisms that establish relative prices amongst goods and services and allocation of limited resources amongst many alternative uses.

Whereas, economists use microeconomic theory to model phenomena observed in the real world, computer scientists use the same theory to let distributed software systems behave in a desired way. Market-based computing is becoming a central paradigm in the design of distributed systems that need to act in complex environments. Market mechanisms provide a way to incentivize parties (in this case software agents), that are not under direct control of a central authority, to behave in a certain way. A microeconomic theory commonly used in MAS is that of general equilibrium. In general equilibrium markets, or exchange markets, all agents respond to the same price, that is determined by searching for the price that balances all demand and supply in the system. From a computational point of view, electronic equilibrium markets are distributed search algorithms aimed at finding the best trade-offs in a multidimensional search space defined by the preferences of all agents participating in the market. The market outcome is *Pareto* optimal, a social optimal outcome for which no other outcome exists that makes one agent better-off without making other agents worse-off.

2.4 PowerMatcher concept

TNO has developed an intelligent distributed coordination technology called *PowerMatcher*. The PowerMatcher is a multi-agent based system that uses electronic exchange markets to coordinate a cluster of devices to match its electricity supply and demand. A multi-agent system is a structured framework for implementing complex, distributed, scalable and open ICT systems in which multiple software agents are interacting in order to reach a system goal. Such a software agent is a self-contained software program that acts as representative of something or someone (in this case a device or an energy demand/supply from the user). A single software agent carries out a specific task. For this task, it uses information from and performs actions in its local environment. It is able to communicate with other entities (agents, systems, humans) for its task. When designed well, the intelligence level of the over-all system is high, while the complexity of individual agents is low. The different PowerMatcher agents and their interactions are shown in Figure 1.

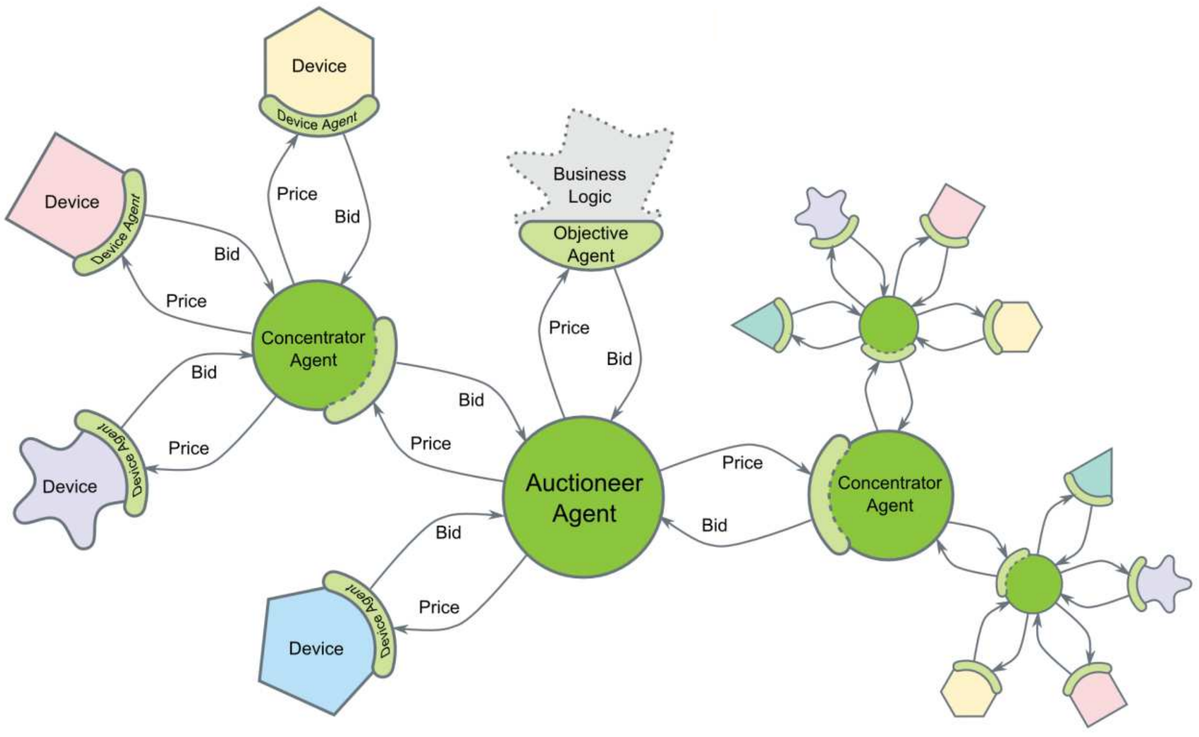


Figure 1: Schematic overview of the PowerMatcher concept.

The only information that is exchanged between the agents and the auctioneer are bids. These bids express to what degree an agent is willing to pay or be paid for a certain amount of electricity. Bids can thus be seen as the priority or willingness of a device to turn itself on or off. For example, a freezer that has almost warmed up is more eager to pay a higher price for its electricity than a freezer that is still cold. Device agents are assumed to be rational, i.e. its behavior follows the basic economic principles. This means that it will be less eager to buy and more eager to sell electricity when prices are higher. Therefore a bid function must always be continuously monotonically decreasing. Furthermore, although prices may only be used as a control signal and not for billing purposes, it is assumed that device agents always bid against their actual (marginal) costs.

"A bid function must always be monotonically decreasing"

Bids are send at irregular (event-based) intervals, i.e. only when an agent's bid has changed. This keeps the communication between PowerMatcher entities to a minimum. The auctioneer collects the bids and calculates the market clearing price. This is the price at which the production and consumption are in equilibrium. The market clearing price is communicated back to the device agents, which react appropriately by either starting to produce or consume electricity, or wait until the market price or device priority (state) changes. The auctioneer is always a passive entity, thus it only acts if it receives new bids that results in a change in price. Actions are triggered at the lowest level, i.e. the device agents, in the PowerMatcher hierarchy, therefore creating a bottom-up approach.

2.5 Agent roles

Within a PowerMatcher cluster the agents are organized into a logical tree. The leafs of this tree are a number of *local device agents* and, optionally, a unique *objective agent*. The root of the tree is formed by the *auctioneer agent*, a unique agent that handles the price forming, i.e. the search for the equilibrium price. In order to obtain scalability, *concentrator agents* can be added to the structure as tree nodes.

- The **device agent** is a representative of a (DER) device. It is a control agent which tries to operate the physical process associated with the device in an economical optimal way. This agent coordinates its actions with all other agents in the cluster by buying or selling the electricity consumed or produced by the device on an electronic market. In order to do so, the agent communicates its latest bid to the auctioneer and receives price updates from the auctioneer. Its own latest bid, together with the current price, determines the amount of power the agent is obliged to produce or consume.
- An **auctioneer agent** performs the price-forming process. The auctioneer aggregates the bids of all agents directly connected to it into one single bid, searches for the equilibrium price and communicates a price update back whenever there is a significant price change. Each PowerMatcher system has only one auctioneer, which is the highest agent in the logical tree.
- **Concentrator agents** are representatives of a sub-cluster of local device agents. It aggregates the market bids of the agents it represents into one bid and communicates this to the auctioneer. In the opposite direction, it passes price updates to the agents in its sub-cluster. This agent uses 'role playing'. On the auctioneer's side it mimics a device agent: sending bid updates to the auctioneer whenever necessary and receiving price updates from the auctioneer. Towards the sub cluster agents directly connected to it, it mimics the auctioneer: receiving bid updates and providing price updates. Therefore, multiple levels of concentrators can exist in a PowerMatcher cluster.
- The **objective agent** gives a cluster its purpose. When the objective agent is absent, the goal of the cluster is to balance itself, i.e., it strives for an equal supply and demand within the cluster itself. Depending on the specific application, the goal of the cluster might be different. If the cluster has to operate as a virtual power plant, for example, it needs to follow a certain externally provided set point schedule. Such an externally imposed objective can be realized by implementing an objective agent. The objective agent interfaces to the business logic behind the specific application.

Although not depicted in Figure 1, a **network officer** exists for each auctioneer/concentrator level in the network, which serves as a 'yellow pages', keeping a list of all agents at its network level and provides the information to device agents to find a nearby matching service (auctioneer/concentrator).

3 Designing a device agent

3.1 Introduction

Designing a device agent can be be a difficult and complex task. It requires knowledge about the device and sometimes the physical process behind it, such as heating of a building. It also requires a different way of thinking than in conventional optimization algorithms. But most importantly, the designer needs to understand what type of flexibility the device has, and how that flexibility can be utilized best. This chapter discusses how trade and bid strategies of a device agent can be designed.

3.2 Bid functions

Device agents communicate with a market through bids and prices. A bid is a function that correlates a commodity price with an amount or volume of the commodity that can be traded. This function can change over time, as the perspective from which the device agent trades on the market can change.

PowerMatcher bids correlate the price for energy (e.g. in €/MWh) with the power (e.g. in kW) a device wants to consume and/or produce. They represent the real-time market position of a device, which can be updated at any given time. Mathematically, a bid *B* can be expressed as a function of price *p*:

$B = B(p)$

while the allocation *a*, i.e. the value of the bid at the market clearing price *p**, is determined following

$$a = B(p^*)$$

Thus, bids do not tell the market how much energy the agent wants to buy or sell at a given price, but merely at what rate it wants to buy or sell. This sounds counter-intuitive, but there are two crucial aspects why PowerMatcher uses power instead of energy.

First, the market system used by PowerMatcher is event-based, meaning the market responds to time-independent 'events' instead of working with pre-defined time-intervals. As energy is an integral product of power and time, the lack of a time component in the event-based market system does not allow for energy trading. Secondly, the electricity system must always be in balance, i.e. supply and demand are in equilibrium. As the electricity system itself is not able to provide a buffer (as opposed to e.g. gas and water infrastructure systems), balancing is a real-time, thus time-independent, activity. Again, this means that balancing needs to be done in units of power and not in units of energy.

Although the commodity volume is traded in units of power, the commodity value or price is expressed in a monetary value per unit of energy, i.e. euros per megawatt hour and not euros per megawatt. To derive the energy consumption (or production) of a device over a given time interval, the (time-dependant) power allocation is integrated over time, i.e.

$$E = \int a(t)dt$$

while the monetary value (costs or earnings) is given by:

$$M = \int a(t)\, p^{*}(t)dt$$

Before looking at properties of bid curves, let’s first consider the following example. Suppose we have a freezer that needs to keep its temperature between -15 and -20 °C. When the temperature is above -15 °C, the refrigerator must turn on, whatever the cost. Likewise, the refrigerator must turn off when a temperature below -20 °C is reached. Let’s assume the freezer turned a short time ago and the temperature is still -19 °C. The device agent understands that this freezer doesn’t have to turn on anytime soon. But on the other hand, there is still some space in the temperature bandwidth to turn the freezer on. So only if the electricity price is very low (let’s say 5 credits per energy unit) the agent will turn the freezer on.

Now what if the market price is higher than this threshold? The freezer stays off and the temperature within slowly increases to -17.5 °C. The time that the freezer can still stay off has now been significantly reduced. As the agent knows that eventually, it will have to turn the freezer on, it is willing to pay a higher price. Maybe the market price is still higher and the temperature in the freezer increases to -16 °C. Now it’s getting critical for the agent. It is getting very close to the point where it will have to buy electricity against any price. But it still has some space to wait. So again, it increases it price threshold. This time it is lucky and the market price happens to be below this threshold: the freezer turns on. Now the game starts again. Every moment, the agent will calculate whether it is worth-a-while to keep the freezer on, or that it should turn it off due to the high market price.

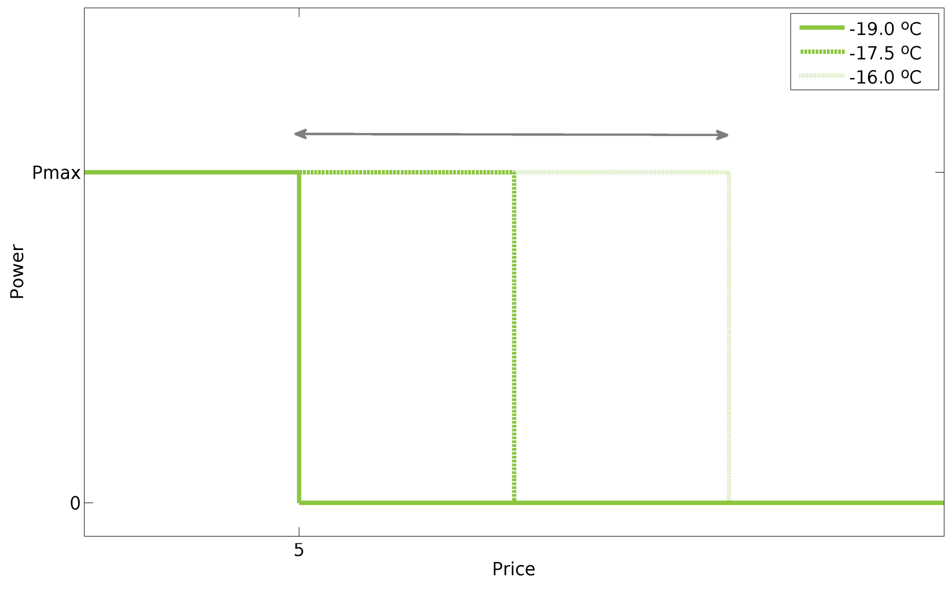


Figure 2: Example of bid curve for a freezer.

This demonstrates the idea of "willingness to pay". The more urgent it gets to turn a device on, the higher the price will be that the agent is willing to pay for the energy needed. This urgency is directly translated in the bid, as the threshold shifts over the horizontal axis, as depicted in Figure 2. Note that although overall, the bid curves are descending, locally, they are constant. To make these bid curves 'monotonically descending', the constant lines are made descending with a very small amount, i.e. B(p) = constant - epsilon x price.

The above example has only taken two evaluations points, at -17.5 °C and -16 °C, but in principle, every arbitrary number evaluation moments can be considered, e.g. every 0.1 °C change in temperature. Such criteria create the events in the event-based market. Has the temperature changed with 0.1 °C, then the current market bid is reconsidered and possibly updated.

Figure 2 showed three bid curves, which were depicted in a graph with the price on the horizontal axes and power consumption on the vertical axis. Within PowerMatcher, this is the general way of depicting bids. However, it must be noted that in economic theory, the commodity volume is usually on the horizontal axis, while the commodity price is on the vertical axis. Furthermore, economic theory expresses the supply of and demand for a commodity in separate bids. The demand bid curve is a descending line with the price on the horizontal axis as there is less demand at higher price, while the supply curve is an ascending line (more supply at higher prices). The intersection of these curves determines the market clearing price and the volume being traded. In PowerMatcher no distinction is made between demand and supply curves. They are one and the same curve, in which a positive power is a demand, a negative power a supply. This is the end-customer ("consumer") perspective of the energy market. Note that utility and grid companies do the opposite. They look from a producer's perspective and use positive production and negative consumption. The producer's perspective is also seen on wholesale markets, such as the day-ahead and balancing market.

The main advantage of using a single bid curve for both supply and demand, is that it can easily be aggregated. An aggregated bid is the representation of the market position of a cluster of devices. The aggregated bid can be calculated by adding the individual bids B_i of devices, i.e.

$$B_{aggr} = \Sigma B_i$$

The market equilibrium price p^* follows from the intersection of the horizontal price axis and aggregated bid at market level, i.e.

$$B_{aggr}(p^*) = 0$$

and is also shown in Figure 3.

It is assumed that device agents always make rational decisions, i.e. at higher prices, they will try to sell more or buy less. Thus, bid curves are and must always be descending (monotonically decreasing).

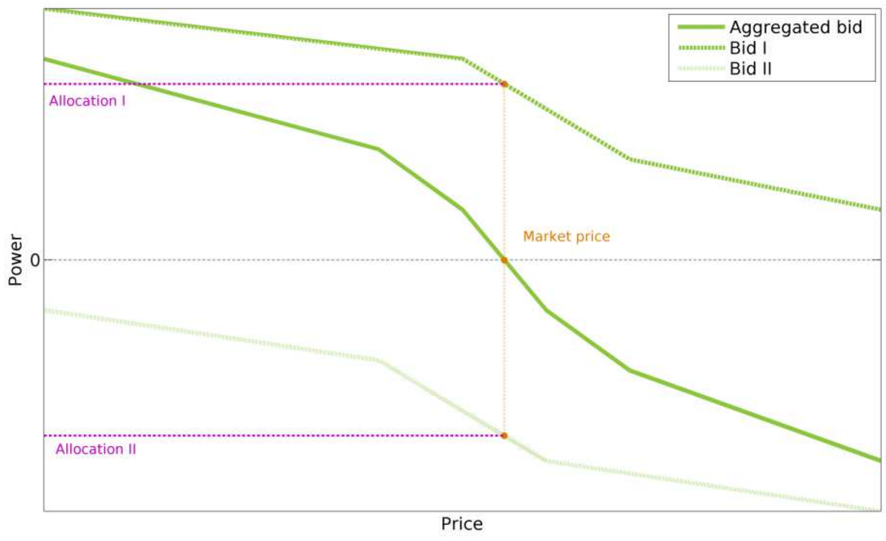


Figure 3: Market equilibrium price is the intersection between horizontal axis and aggregated bid

3.3 Flexibility

A device has flexibility if it is capable of shifting its production or consumption of energy in time. This shift is bounded by two constraints. First, the comfort of the end-user cannot be infringed. The device must operate within the comfort boundaries, such as a temperature range or a time frame in which certain functions (e.g. washing) must be completed. Secondly, the total amount of energy production or consumption remains the same as the end-users comfort demand doesn't change. For example, a washing machine needs 1 kWh for a laundry program. The flexibility of the washing machine is that it can be started at 15:00 or 16:00, but the amount of energy it needs to complete the program stays the same.

Flexibility comes in different shapes and sizes. For example, the freezer example in the previous section used its thermal capacity to decouple the demand for cold from the demand for electricity, while constrained by an upper and lower temperature limit. Thus its flexibility lies in its thermal capacity. The washing machine on the other hand, has a deadline at which the laundry must be done. This gives the washing machine a time window in which it can shift the demand. The constraints for the washing machine are the length of the laundry program and the remaining time to the deadline.

These examples demonstrate that the freezer has a different type of flexibility than the washing machine, using a buffer respectively a time window. In total, five different types of flexibility can be identified, each with their own characteristics, as shown in Table 1.

Table 1: Different categories of device flexibility.

Type	Characteristics	Examples

Inflexible	• Situated in a "must-run" or "must-off" state. • Direct control by the end-user. • Unable to shift consumption or production of energy to another time	*Lighting, television, solar panels*
Open flexibility	• Can provide flexibility at any moment • Flexibility is available for relative long or infinite time.	*Diesel generator, hydro power*
Buffering flexibility	• Device consumes/produces two or more commodities, one of them being electricity. The consumption/production of the commodities are coupled. • Flexibility is provided by buffering one the commodity or commodities that is not electricity, e.g. heat, cold, gas or CO_2 • Indirect control by the end-user (e.g. thermostat) • Flexibility is bounded by buffer limitations (e.g. max/min buffer temperature)	*HVAC (heating, cooling and ventilation)*
Time-window flexibility	• Has a time deadline when an operation must be completed. • Flexibility is provided by the time window available with respect to the time needed for the device operation.	*Washing machine, dish washer, dryer,*
Storage	• The primary function of the device is to store electricity. • Flexibility is provided by storing electricity temporarily and supply it at a later moment.	*Batteries, capacitors, flywheels*

These categories should be seen as stereotypes. Any flexible device has characteristics that fit within these categories. The categories are however not mutually exclusive, i.e. the flexible device may fit in more than one category. Take For example, the increasingly more popular electric vehicle. It temporarily can store electric energy in its battery, which would make fit it in the 'storage' category. However, the battery must be fully charged when the driver wants to use the car. So it also needs to be charged within a given time frame, making it better fit in the 'time-window' category. In which category it fits best, depends on the size of the time window with respect to the time that is required to charge the vehicle's battery. If the time window is relatively long, it behaves like storage. Otherwise its flexibility fits in the time window category. A device agent should always be able to adapt to changes in the type of flexibility.

Another example of category overlaps are renewable energy sources. In general, renewables are inflexible. They provide electricity when natural resources, like wind, solar energy and water, are available. Less natural resources means less electricity being produced. However, there are ways to partially control RES devices, thus creating some flexibility. A dam can be built in a river to buffer the water for a turbine. The blades of a wind turbine can be tilted to reduce its power output. Although the flexibility that is provided by these measures is limited, the control action itself is unconstraint: the turbine blades can be tilted perpetually. Thus, RES units are both inflexible and unconstraint (open) flexible.

3.4 Trade strategy

In the freezer example in section 3.1, only a rough outline was given on how a device agent can bid on the market. It was made clear that the freezer would be willing to pay more if its temperature was higher, but how much more? And what should the freezer have bid when its temperature was -19 degrees Celsius? And what if the freezer was an electric car? When should it charge its battery? In other words, information from the physical processes needs to be converted to an actual bid curve. There are two steps needed in achieving this. First, the process state information needs to be mapped onto monetary values, which is called **trade strategy**. Secondly, the monetary values, thresholds and device properties lead to the construction of a bid function or bid curve. This is the **bid strategy**, which will be discussed in the next section. In other words, the trade strategy determines the willingness to pay of the device, while the bid strategy determines the shape of the bid that is send by the device agent to the auctioneer.

Let's start looking at a diesel generator with an electrical capacity of 1 kW. It can only be turned on or off, i.e. it either provides 1 kW or nothing at all. It's efficiency (fuel to electricity conversion) is 25% and fuel costs the diesel price is 0.10 €/kWh. At what price will the generator be willing to turn on? It will take four kWh of diesel to create one kWh of electricity, so the value of electricity is four times higher than that of the diesel, making it 0.40 €/kWh. If the electricity price on the market is higher than this value, the generator makes a profit. Is the market price lower, the generator makes a loss. Thus, the 0.40 €/kWh is a monetary threshold for the generator. However, this threshold is not dependent on the state of the physical process (assuming the efficiency stays the same). This is usually not the case for other flexible devices.

Let's get back to the example of the freezer in section 3.2. It was concluded that the freezer was willing to pay more if its temperature was higher. Thus, it's threshold is a function of the temperature of the buffer with respect to its boundary values, i.e.

$$p_{thres} = F(T, T_{max}, T_{min})$$

For every device and process, such a process parameter dependent function can be derived. It can even be generalized towards the different flexibility categories, as is shown in Table 2.

Table 2: Flexibility category and the process parameters that could be used to derive a monetary threshold.

Type	Dependent on
Inflexible	• N/A
Open flexibility	• Fuel price • Energy conversion efficiency
Buffering flexibility	• Current buffer level • Maximum and minimum buffer levels

Time-window flexibility	• Time required to perform the device operation • Time left until the device operation must be completed
Storage	• Current buffer level • Maximum and minimum buffer levels

This table merely shows the most basic (process) parameters used to determine a threshold. There are many additional parameters that could be used. For example, the amount of food stored in a refrigerator or freezer determines its thermal capacity (i.e. buffer size) and is thus variable. Another example, the forecasted heat demand of a building can determine whether a critical situation may exist if the thermal buffer of a heating device (CHP or heat pump) would be empty. After all, if it is a hot summer day, there is no need to fill up the buffer, but on a cold winter day, you wouldn't want to run out of heat. So the willingness to pay of a building heating device can also be season dependent. A concrete example of how the monetary threshold can be calculated is given in the next chapter.

3.5 Bid strategy

Bid strategies use the monetary value of the process, as determined by the trade strategy, to create a bid function. These strategies don't have to be complex. Devices that can only turn on or off, have only one monetary threshold (cost price). At one side of this threshold the device is on, at the others side it is off, i.e.

	Consumer	Producer
ON	$p^* < p_{treshold}$	$p^* > p_{treshold}$
OFF	$p^* > p_{treshold}$	$p^* < p_{treshold}$

However, some devices may have more difficult strategies. Take for example a battery, which can both produce and consume electricity. So it has at least two thresholds: one for charging and one for discharging. It even goes a step further. The monetary thresholds will be dependent on the cycle efficiency of the battery, which on its turn is a function of the charging or discharging power. The lower the power, the higher the efficiency. Figure 4 shows how the bid function for a battery could look like, in which *e* and *c* are the two thresholds for charging respectively discharging with nominal power.

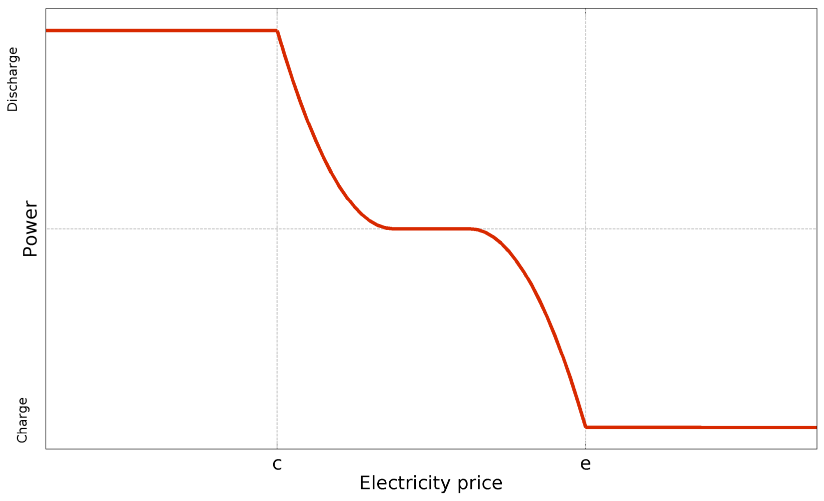


Figure 4: Bid curve created by a battery device agent.

4 Example: Combined heat and power plant

4.1 Introduction

This chapter explains the design of a device agent that represents a combined heat and power unit (CHP). The design and algorithms presented have been successfully used by TNO in the recent field trials and simulations. A CHP is a device that converts a fuel into electricity and heat. The heat is of such quality (i.e. high temperature) that it can and is effectively used. As CHPs have a useful purpose for their heat, they are much more efficient that conventional power plants, which discard the by-product heat. Most electricity generating devices can be modified to function as a CHP and are either based on a sterling engine, gas turbine or diesel engine. Commonly, they use natural gas, biogas or (bio-)diesel as a fuel.

CHP are commonly found in sectors that require large amounts of heat of relative low temperatures (60-90 °C), such as greenhouses. Since 2009, CHPs are also commercially available for providing spatial and tap water heating in households, the so called micro-CHPs. Hence, the size of CHPs varies a lot and ranges from 1 kW_{el} up to 50 MW_{el}.

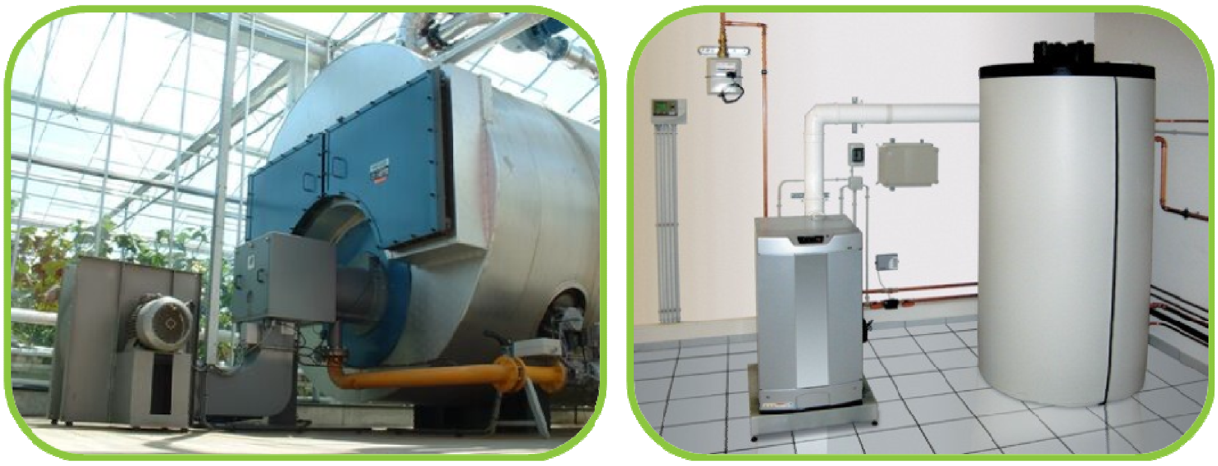


Figure 5: Medium sized CHP in a greenhouse (left) and a micro-CHP for domestic heating (right).

In principle, a CHP can be operated in three ways.

- **Electricity-driven**, where a CHP primarily functions as a conventional power plant, following the (national) demand for electricity. Heat is a by-product that is fed into a district heating network.
- **Fuel-driven**, where a CHP is combined with a biofuel installation, such as a digester. The digester uses an bacteria-based anaerobic digestion process to convert biomass or sewer sludge into gas with a high methane concentration. For optimal process speed, the temperature in the digester must be about 30-50 °C, depending on the bacteria species used. The CHP produces more than sufficient heat to maintain this temperature. Thus, the digester and CHP form a closed operational loop, with electricity as the main export product.

- **Heat-driven**, where a CHP can be a replacement of a conventional heater to cover a certain heat demand, while electricity is considered to be a by-product. As electricity is relatively expensive, a CHP can be financially more interesting that a conventional heater. This business model is seen often in the greenhouse sector, which have all year round demand for heat. If there is no demand for heat, the CHP is not operational.

Heat driven CHPs are usually complemented with a form of thermal storage, such as a hot water buffer. The buffer partially disconnect the demand for heat and production of electricity by the CHP. This provides flexibility to optimize CHP operation, e.g. longer continuous running times or shifting electricity production to periods with high prices. This chapter will describe the design of a trade and bid strategy for a simple heat-driven CHP and the implementation of the device agent that can represent the CHP.

4.2 Modeling

The association of components needed to model a heat-driven CHP is shown in Figure 6.

Figure 6: Interaction between components of a CHP heating system.

The time scale of the CHP system model is in the order of 1 – 10 minutes. It is therefore not necessary to model the dynamic behavior of the CHP. This behavior occurs mainly during start-up and shutdown phases when all kinds of non-linear processes are taken place. Furthermore, as the focus is on electricity trading, it is sufficient to model energy flows only, so temperatures are not taken into account. It is assumed that the temperature of the heat produced by the CHP is always sufficient high enough to be stored in the heat buffer. Based on these assumptions, the CHP component can be modeled based on the following three parameters:

Quantity	Units	Description
η_{el}	%	The electric efficiency of the CHP, i.e. the ratio between the amount of electrical energy output and fuel energy input.
η_{th}	%	The thermal efficiency of the CHP, i.e. the ratio between the amount of thermal energy output and fuel energy input.
S_{el}	watt	The nominal electric power output of the CHP.

Optionally, the nominal thermal power output could be used instead of the nominal electrical power output. If the CHP is turned on it produces S_{el} electricity and $\eta_{th}\frac{S_{el}}{\square}\eta_{el}$ of heat.

Energy buffers, whether they store heat, electricity or fuel, can be characterized by a single parameter: the nominal (i.e. maximum) energy capacity Q_0. If the buffer is zero, the energy capacity is zero. If the buffer is full, it equals the nominal energy capacity. For separate heat buffers, the nominal energy capacity can be calculated using the following parameters:

Quantity	Units	Description
V	m^3	The volume of the buffer.
ρ	kg/m^3	The density of the material (usually a fluid) of the buffer.
c_p	$J\ kg^{-1}\ K^{-1}$	The specific energy density of the buffer material.
T_{min}	°C	The minimum allowable temperature of the buffer, i.e. the temperature at which the heat is unusable.
T_{max}	°C	The maximum allowable temperature of the buffer.

Most common material used for a heat buffer is water, which has a specific energy density of 4180 J kg^{-1} K^{-1}. The nominal energy capacity can then be calculated following

$$Q_0 = c_p \rho V (T_{max} - T_{min})$$

and the energy capacity of the storage for a given temperature *T* using:

$$Q = c_p \rho V (T - T_{min})$$

A temperature independent indicator to express how much the energy buffer is filled, is the state of charge or SOC. The state of charge is 0 when the buffer is empty and 1 (or 100%) if the buffer is full. For the above heat buffer, the state of charge of the storage can be calculated following:

$$SOC = \frac{Q}{Q_0} = \frac{(T - T_{min})}{(T_{max} - T_{min})}$$

For a given time step dt , the change in the energy capacity in the buffer can be expressed as

$$dQ = (\dot{q}_{th} - \dot{q}_{demand})\, dt$$

or in terms of the change in state-of-charge:

$$d(SOC) = \frac{\dot{q}_{th} - \dot{q}_{demand}}{Q_0}\, dt$$

where $\dot{q}_{th}$ is the thermal power production of the CHP and $\dot{q}_{demand}$ the thermal power demand. It is assumed that the heat demand is known (data series) and does not need to be modeled.

4.3 Trading strategy

Like every other device, the CHP wants to minimize its costs and maximize its financial gains. If we consider only the revenues and expenses made per unit of fuel, disregarding fixed and maintenance costs, then the gross profit *P* (per unit of fuel) can be expressed as:

$$P = \eta_{el} p_{el} + \eta_{th} p_{heat} - p_{fuel}$$

where P_{el}, P_{heat} and P_{fuel} are the price for respectively electricity, heat and fuel. A CHP makes profit as long as P > 0. In our experiments it is assumed that only the electricity price is variable. All other prices are considered to be constant. Based on this assumption and the operation modes, the minimum electricity price $\hat{p}_{el}$ can be calculated that is required to run the CHP profitable:

$$\hat{p}_{el} = \frac{p_{fuel} - \eta_{th} p_{heat}}{\eta_{el}}$$

If the heat would have no value or is being discarded, this equation can be simplified to

$$\hat{p}_{el} = \frac{p_{fuel}}{\eta_{el}}$$

If the CHP is operated in fuel driven or electricity driven mode, it is assumed that there are no constraints on the delivery of heat.The price for heat is assumed to be known and fixed. In the heating district of Amsterdam, heat has a value of about 20 €/GJ.

However, if the CHP is heat demand driven, the value of the heat needs to be quantified. This may be difficult as the value strongly depends on the comfort requirements of the end customer. A user-independent approach estimating the heat value is to consider the most common alternative for heat production. For a CHP, this usually is a gas fired heater or sometimes an electrical (resistance) heating. The costs to produce heat with the alternatives can be easily determined. In case of a gas fired heater, the minimum electricity price becomes:

$$\hat{p}_{el} = \frac{p_{fuel}}{\eta_{el}} - \frac{\eta_{th}}{\eta_{el}} \frac{p_{alt}}{\eta_{alt}}$$

where η_{alt} is the efficiency and P_{alt} the price of fuel used by the gas fired heater. In case of electrical heating, the equation becomes:

$$\hat{p}_{el} = p_{fuel} \left[\frac{\eta_{th}}{\eta_{alt}} + \eta_{el} \right]^{-1}$$

with η_{alt} efficiency of the electric heater. Note that the fuel price of an electric heater is the electricity price, and thus ends up at the left side of the equation.

When heat driven, the amount of electricity that is produced is limited and depends on the heat demand. To maximize profit of the CHP, electricity should be produced in times with the highest possible price. But these prices are not known beforehand, so planning is not option. Heat storage partly decouples electricity and heat demand, allowing the shift of electricity production to peak prices. But as the storage is limited in size and heat is the primary product of the CHP, a trade-off between heat demand and electricity prices needs to be found.

A solution is to map electricity prices onto the state of charge (SOC) of the thermal buffer. The mapping is graphically shown in Figure 7. If the buffer is empty, heat must be produced. A CHP is willing to do so, if the electricity price is above $\hat{p}_{el}$. If not, a conventional heater (ALT) is cheaper to use. If the buffer is full, no more heat can be stored and must be discarded (DUMP). The heat has no value and the CHP can only operate as a generator. The electricity price at which a generator is profitable was given as:

$$\hat{p}_{dump} = \frac{p_{fuel}}{\eta_{el}}$$

From an energy efficiency point of view, dumping heat is not a good thing. Therefore, it is also possible to turn the CHP off when the buffer is full, even if the price for electricity would be profitable.

The electricity price at which the CHP can and should run when the buffer is partly filled can be calculated in many ways. A simple solution is to use a linear mapping. In earlier PowerMatcher field demonstrations, linear mapping proved to be a good balance between algorithm implementation time and optimal trading strategy. Other high or low risk mappings can be based on heat demand forecasts, expected price fluctuations and user requirements, but are therefore performance-heavy and more prone to errors. To prevent on/off flipping of the CHP, the mapping could be implemented as a hysteresis loop: one curve to decide when the CHP turns on and one curve to turn it off.

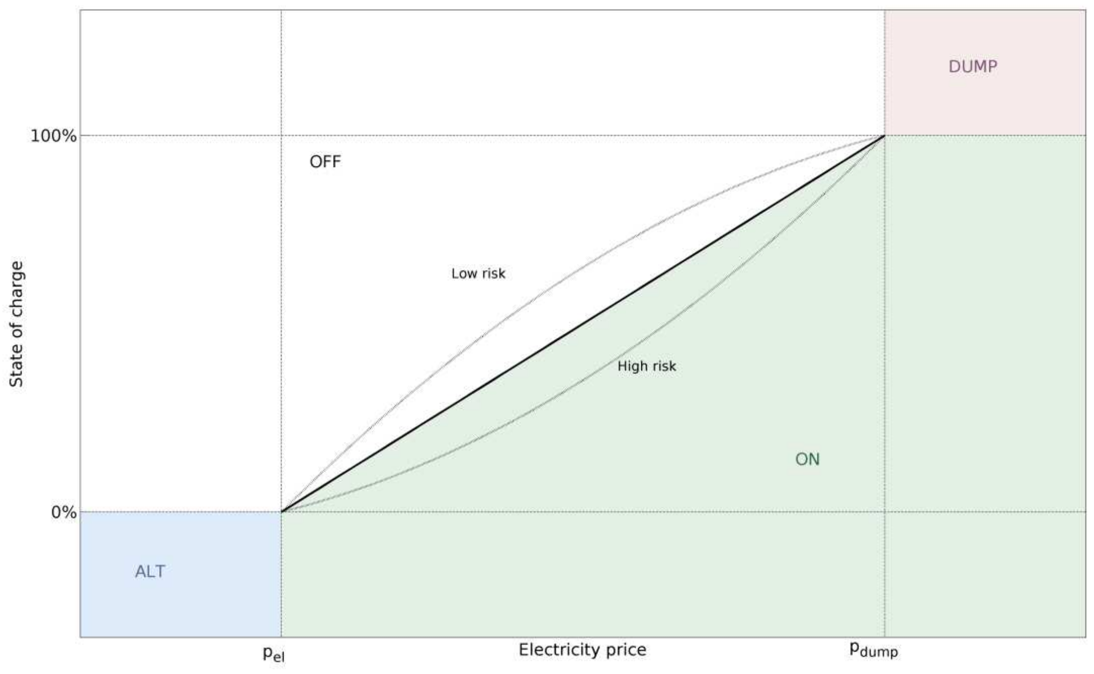


Figure 7: Mapping of state of charge onto the electricity price.

In case there are two or more identical CHPs using the same buffer (i.e. cascading), different risk profiles should be used. This prevents that the CHPs always turn on simultaneously, which reduces the running time of the CHPs, and thus their efficiencies.

4.4 Bidding strategy

The bidding strategy is based on the trading strategy. Assuming the CHP only knows an on and off state, than the bidding curve *B* is described by:

$$B(p_{el}) = \begin{cases} 0 & p_{el} \leq p_{map} \\ S_{el} & p_{el} > p_{map} \end{cases}$$

where S_{el} is the electric power of the CHP and p_{map} the price following from the mapping at the current state of charge of the buffer. If the CHP is fuel or electricity driven, $p_{map} = \hat{p}_{el}$. Thus the bidding curve is a step function.

Some CHP's are able to modulate their output power, i.e. they can be partially loaded. However, at part load, the electric efficiency is usually lower and the thermal efficiency the same or higher. From the equations in the previous section, it can be derived that the minimum electricity price $\hat{p}_{el}$ at part load is always higher than at nominal load. This means that operational costs are higher for part load than for full load. From a trading perspective, it is therefore illogical to offer part loading to the market, because of two reasons. First, the threshold when the CHP can run profitable ($\hat{p}_{el}$) goes up, making the CHP less competitive in the market. Secondly, less energy is produced per unit of time, and thus a lower turn-over is made.

4.5 Other devices

This chapter only has discussed the trade and bid strategy of one specific device (the CHP). However, the strategies used, specifically, the mapping of the state-of-charge onto an electricity price (i.e. Figure 7) can be used more widely. TNO has used similar strategies for a variety of devices that have buffering flexibility, such as refrigerators, freezers, air conditioners and heat pumps. With some adjustments, it can even be used for devices with electrical storage flexibility.

5 Writing a device agent

5.1 Background

This chapter describes how a device agent can be written with the Java version of the PowerMatcher Core[1] and PowerMatcher Extensions frameworks. **It should be noted that this section may change in later releases as a result of further development of the frameworks.** This background section will explain the architecture of a device agent and its associated interfaces. This section does not provide step-by-step programming instructions. Experienced Java developers are assumed to use this section as the starting point for exploring the referenced template source code.

A PowerMatcher system consists of software agents that communicate bid and price updates as described in section 2. The communication between agents in PowerMatcher systems is based on a layered model, as shown in Figure 8.

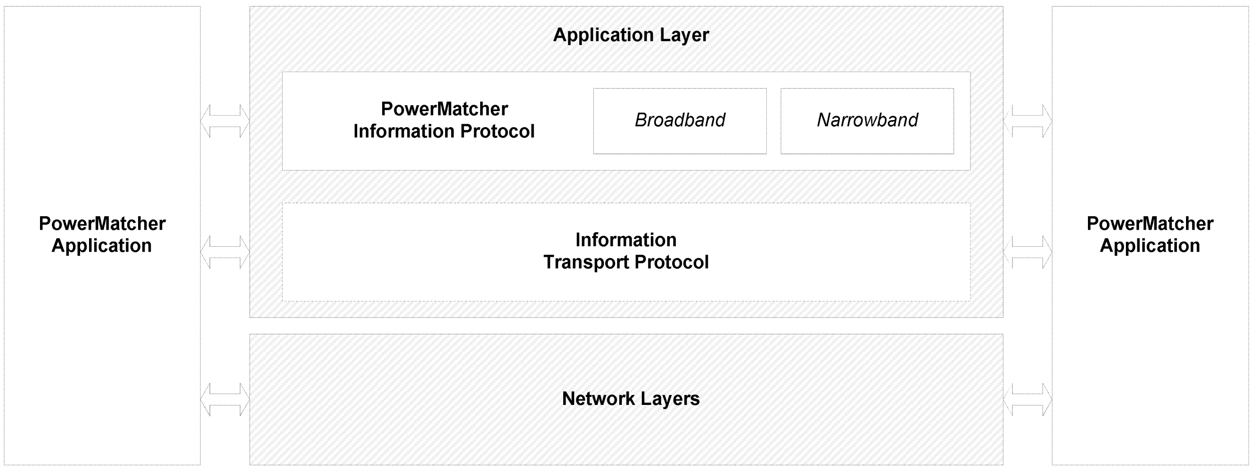


Figure 8: Protocol stack

In this layered model, an Agent implementation can be subdivided into three aspects, as shown in Figure 9.

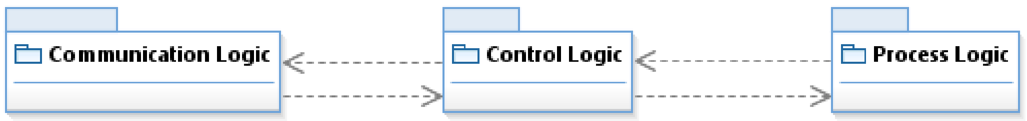


Figure 9: Agent decomposition

[1] *PowerMatcher Core* comprises the reference implementation, *PowerMatcher Extension* provides add-on functionality that can be used as-is.

The ‘Communication logic’ aspect controls the communications with the PowerMatcher entities. This includes both the PowerMatcher Information Protocol as well as, for example, a distributed energy resource (DER) or device interfacing protocol. In a PowerMatcher cluster Agents and Matchers are logically organized in a communication hierarchy as depicted in Figure 10.

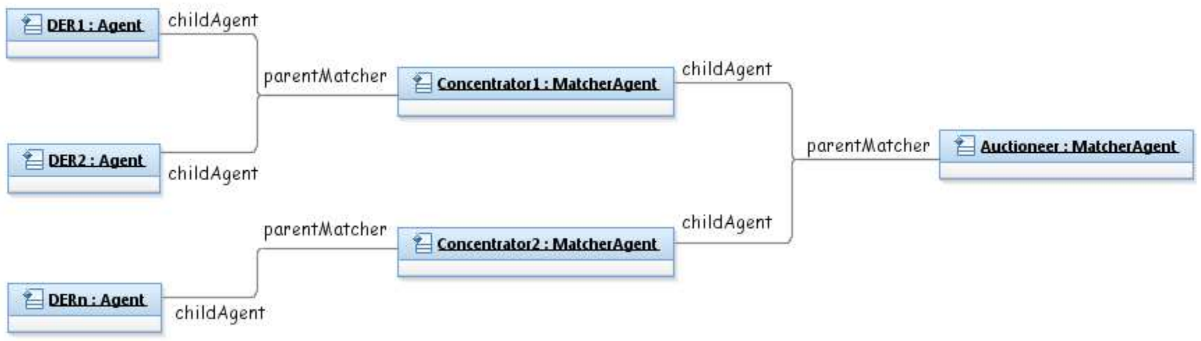


Figure 10: Interfaces in the logical Agent hierarchy

The communication logic allows Agents to be decoupled and deployed in a distributed environment, and is implemented in the form of layered **Adapter** components. PowerMatcher Core provides standard Adapter components for the PowerMatcher Information Protocol[1], and the framework for implementing custom Adapters for device interfacing. The example of Figure 12 shows a Concentrator Agent that is configured with a PowerMatcher Information Protocol Adapter component for a direct connection between agent and matcher interfaces. A direct connection is supported when the agent and the matcher are deployed in the same PowerMatcher runtime.

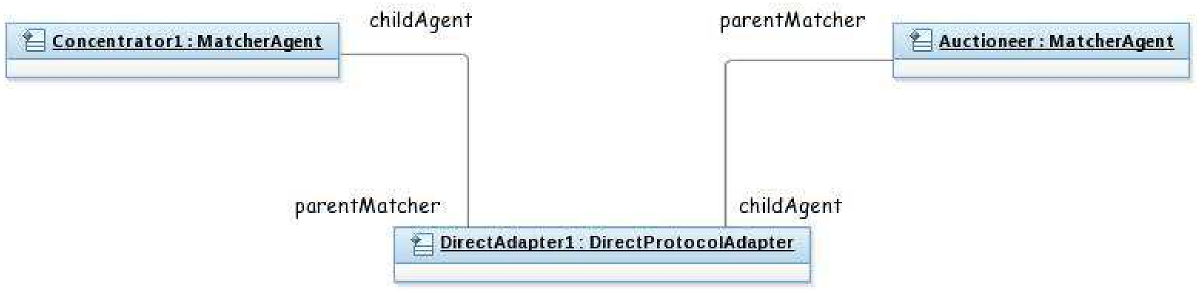
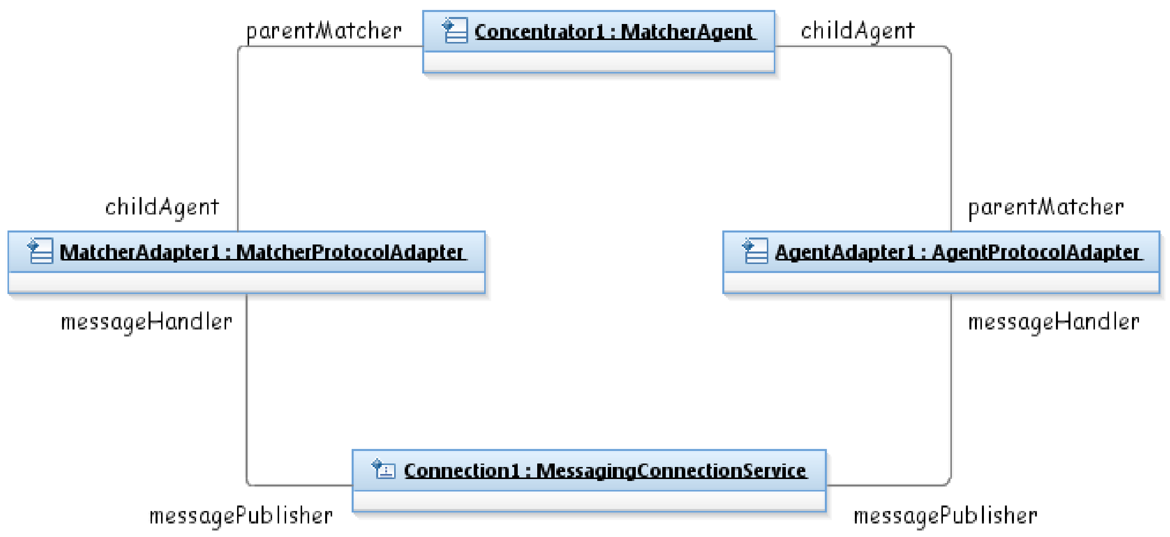

Concentrator1 : MatcherAgent
childAgent
parentMatcher
Auctioneer : MatcherAgent
parentMatcher
childAgent
DirectAdapter1 : DirectProtocolAdapter

Figure 11: An Agent connected to a Matcher via an adapter for a direct connection

When agent and matcher are not deployed in the same runtime, the agent and the matcher exchange price and bid updates via a messaging protocol. The example of Figure 12 shows a Concentrator Agent that is configured with PowerMatcher Information Protocol Adapter messaging components for the matcher and agent interface of the Concentrator, which are in turn configured with a publish-subscribe messaging Information Transport Protocol Adapter component for the MQ Telemetry Transport protocol[2]



parentMatcher
Concentrator1 : MatcherAgent
childAgent
childAgent
MatcherAdapter1 : MatcherProtocolAdapter
parentMatcher
AgentAdapter1 : AgentProtocolAdapter
messageHandler
messageHandler
Connection1 : MessagingConnectionService
messagePublisher
messagePublisher

Figure 12: An Agent connected to a Matcher via messaging adapters

The ‘Controller logic’ aspect controls the creation of bids and the handling of altering commodity prices. This aspect is provided by the PowerMatcher framework that is the basis for implementing the Agent.

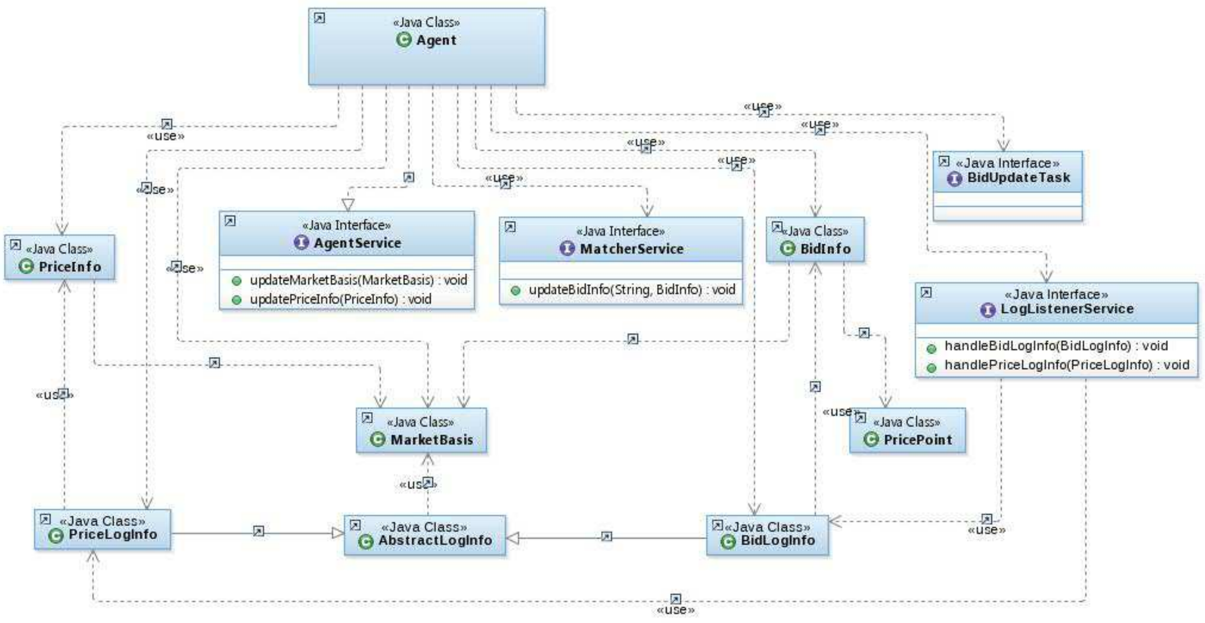
The ‘Process logic’ aspect controls the actual connected device or simulation. This aspect is implemented by the agent developer in the DER Agent and Adapter to interface with the DER.

Besides the frameworks provided by PowerMatcher Core, template projects to jump-start the development of a new Agent and Adapter are included.

In the following subsections, the class diagram of the framework supporting the control aspect is shown in Figure 13, and the process aspect in Figure 14. Finally, configuration of Agent and Adapter components is a common aspect shown in Figure 15, and after that the supply-demand matching interaction sequence is explained.

5.1.1 The Agent framework with the MarketBasis, PriceInfo and BidInfo classes

The framework classes supporting the control aspect are shown in Figure 13 and described in the table following it. These are the classes one primarily works with when implementing a new agent.



«Java Class» Agent
«Java Interface» AgentService
updateMarketBasis(MarketBasis) : void
updatePriceInfo(PriceInfo) : void
«Java Interface» MatcherService
updateBidInfo(String, BidInfo) : void
«Java Class» BidInfo
«Java Interface» BidUpdateTask
«Java Interface» LogListenerService
handleBidLogInfo(BidLogInfo) : void
handlePriceLogInfo(PriceLogInfo) : void
«Java Class» PriceInfo
«Java Class» MarketBasis
«Java Class» PricePoint
«Java Class» PriceLogInfo
«Java Class» AbstractLogInfo
«Java Class» BidLogInfo
«use»

Figure 13: Class diagram for the framework supporting the Agent's control aspect

Class/Interface	Description
AgentService	The interface implemented by an agent to receive market basis and price updates from the parent matcher. The parent matcher (concentrator or auctioneer) uses this interface to publish market basis and price updates.
MatcherService	The interface used by an agent to publish bids to the parent matcher. This interface is implemented by the parent matcher to receive bids from the agents it serves.
Agent	Framework class that is used as the super class for implementing an agent. This class implements the control logic for receiving and processing market basis and price updates, and for publishing bid updates. Examples of agents are provided in the template projects.
MarketBasis	An immutable data class that defines a market basis by commodity, currency, minimum price, maximum price, number of steps and significance. The agent receives market basis updates from its parent matcher.
PriceInfo	An immutable data class that defines a market price by market basis and amount in the currency defined by the market basis. The agent receives market price updates from its parent matcher.
BidInfo	An immutable data class that defines a bid curve published by the agent to its parent matcher. A bid can be represented either in the form of a line curve defined by price points, or by an array specifying the demand for each price step in the market basis.

PricePoint	A data class that defines a price point in a bid curve. A price point specifies the demand at a specific price.
PriceLogInfo	A data class that defines the logging information for logging a market price. The agent can log the market price it receives. A matcher can log the market price it publishes and receives.
BidLogInfo	A data class that defines the logging information for logging a bid. An agent can log the bid it publishes. A matcher can log the bids it receives, the aggregation of the bids it receives and the aggregated, transformed, bid it publishes.

5.1.2 The Connector and Adapter framework

The Connector and Adapter framework shown in Figure 14 supports the process aspect to be implemented in the form of exchangeable plug & play adapter components.

- An Adapter connects to a Connector of another component to provide a particular implementation of, for example a PowerMatcher protocol or device protocol, or of an algorithm.
- A component can implement one or more ConnectorService interfaces to define the connectors that it provides.

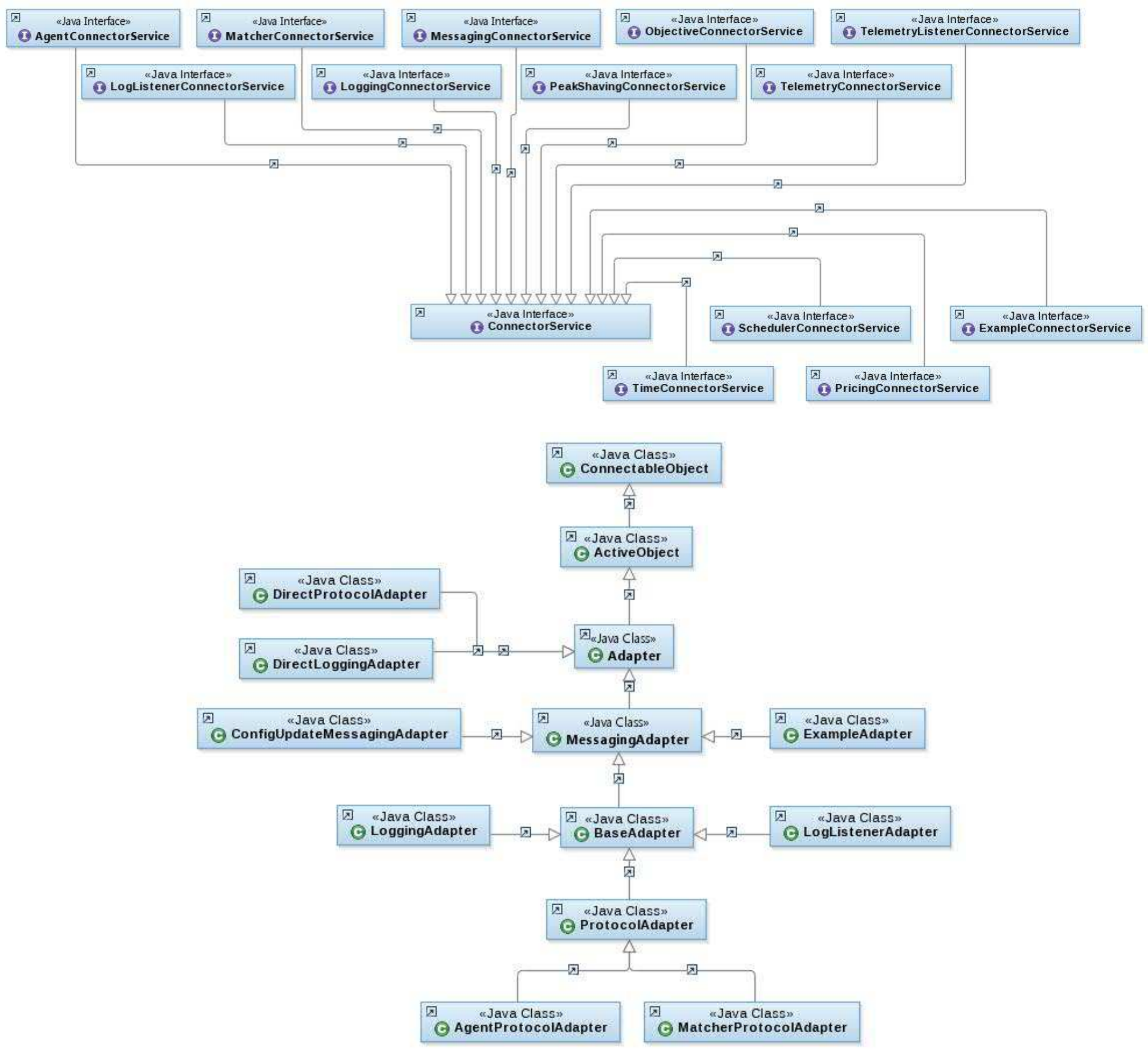

«Java Interface» AgentConnectorService
«Java Interface» MatcherConnectorService
«Java Interface» MessagingConnectorService
«Java Interface» ObjectiveConnectorService
«Java Interface» TelemetryListenerConnectorService
«Java Interface» LogListenerConnectorService
«Java Interface» LoggingConnectorService
«Java Interface» PeakShavingConnectorService
«Java Interface» TelemetryConnectorService
«Java Interface» ConnectorService
«Java Interface» SchedulerConnectorService
«Java Interface» ExampleConnectorService
«Java Interface» TimeConnectorService
«Java Interface» PricingConnectorService
«Java Class» ConnectableObject
«Java Class» ActiveObject
«Java Class» DirectProtocolAdapter
«Java Class» DirectLoggingAdapter
«Java Class» Adapter
«Java Class» ConfigUpdateMessagingAdapter
«Java Class» MessagingAdapter
«Java Class» ExampleAdapter
«Java Class» LoggingAdapter
«Java Class» BaseAdapter
«Java Class» LogListenerAdapter
«Java Class» ProtocolAdapter
«Java Class» AgentProtocolAdapter
«Java Class» MatcherProtocolAdapter

Figure 14: Class diagram for the Adapter framework supporting the Agent's process aspect

Class/Interface	Description
AdapterService	This interface defines the lifecycle methods for identifying, binding and unbinding adapters from connectors. It is used by the framework for starting up and connecting to adapters.
Adapter	The framework class that is used as the super class to implement new adapters. An adapter connects to a component to provide a particular implementation of a service to a component, like for example protocol and device interfacing. All adapters are created from the *AdapterFactory* for a specific type of adapter.
MessagingAdapter	The framework class that is used as the super class to implement

	new messaging adapters. A messaging adapter is an adapter that uses publish-subscribe messaging to interface with its environment. *MessagingAdapter* itself implements *MessagingConnectorService*, which in turn is used for the connection with a specific publish-subscribe broker, like for example for the MQTT protocol. An example of a messaging adapter is provided in a template project.
ConnectorService	The super interface for defining connectors. A component that provides a connector implements a sub interface of Connector Service. An agent always provides an agent connector, and optionally provides custom connectors, for example for device interfacing or plug & play algorithms.
AgentConnectorService	The connector service implemented by the *Agent* framework class to connect to its *AgentService* interface. The adapter for the PowerMatcher Information Protocol uses this connector to bind to an agent.
MatcherConnectorService	The connector service implemented by the *MatcherAgent*[1] framework class to connect to its *MatcherService* interface. The adapter for the PowerMatcher Information Protocol uses this connector to bind to a matcher.
MessagingConnectorService	The connector service implemented by the *MessagingAdapter* framework class
TimeConnectorService *SchedulerConnectorService*	The connector services implemented by the *TimeAdapter* and *SchedulerAdapter* classes. The *TimeAdapter* provides real or simulated time services. The *SchedulerAdapter* provides a thread pool based scheduler operating in real or simulated time.
LoggingConnectorService *LogListenerConnectorService*	The *LoggingConnectorService* is used to provide a service for logging PowerMatcher events to a log listener. The *LogListenerConnectorService* is used to connect to the interface of a logging component that receives PowerMatcher log events.
TelemetryConnectorService *TelemetryListenerConnectorService*	The *TelemetryConnectorService* is an extension service used to provide a service for logging monitoring and status events to a telemetry listener. The *TelemetryListenerConnectorService* is used to connect to the interface of a logging component that receives telemetry events.

[1] The *MatcherAgent* class is not covered in this document, but it is the framework class that is the super class for implementing PowerMatcher concentrator and auctioneer classes.

PricingConnectorService *ObjectiveConnectorService* *PeakShavingConnectorService*	The *PricingConnectorService* is an optional connector of the *Auctioneer* to plug in a custom algorithm to calculate the equilibrium price. The *ObjectiveConnectorService* is an optional connector of the *ObjectiveAgent* to inject a dynamically calculated objective bid into the cluster. The *PeakShavingConnectorService* is used to provide dynamic peak shaving constraints to a peak shaving concentrator.
ExampleConnectorService	To interface with external devices, possibly in generic way based on a generic device model, an agent will implement a device or model specific connector service. This is the point in the hierarchy where the custom connector types are defined. An example is provided in the template projects.

5.1.3 The configuration framework

Configuration of Agent and Adapter components is a common aspect supported by the classes and interfaces shown in Figure 15, and explained in the table following it.

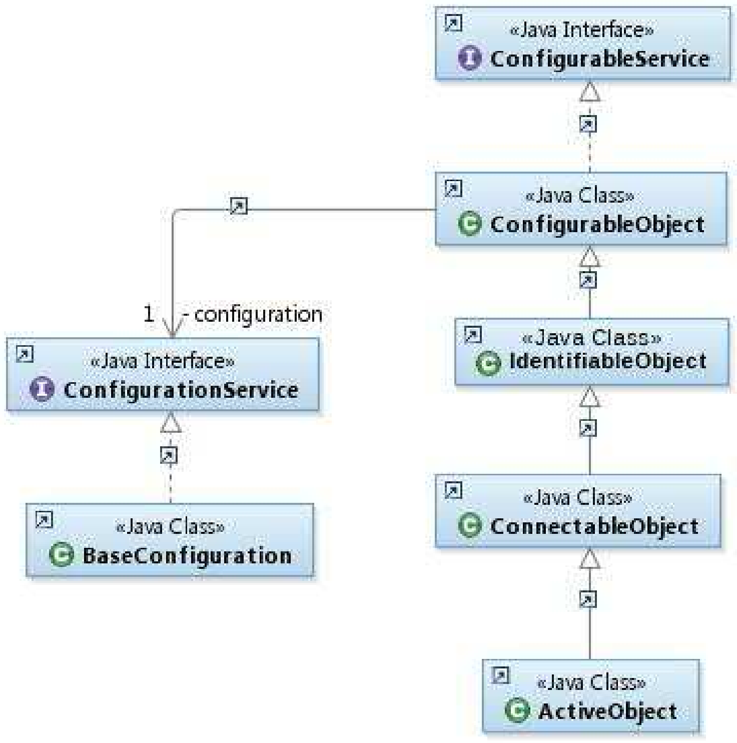

«Java Interface» ConfigurableService
«Java Class» ConfigurableObject
1
- configuration
«Java Interface» ConfigurationService
«Java Class» IdentifiableObject
«Java Class» BaseConfiguration
«Java Class» ConnectableObject
«Java Class» ActiveObject

Figure 15: Class diagram for Agent and Adapter configuration

Class/Interface	Description

ConfigurationService	This interface provides typed access to a set of configuration properties, and it is used to configure agent and adapter components. The source of a configuration can be system properties, property files, configurations defined in the OSGi Configuration Admin service, etc.
BaseConfiguration	This class provides a *ConfigurationService* based on a *java.util.Map*. A *BaseConfiguration* optionally delegates to a parent configuration to support global properties in a configuration hierarchy.
ConfigurableService	This is the interface that is implemented by objects that can be configured by injecting a *ConfigurationService* into it.
ConfigurableObject	A framework class that can be used as super class for configurable objects. It implements *ConfigurableService* and provides protected utility methods for types access to the configuration properties.
ConnectableObject	The base class that serves as the super class for all PowerMatcher agents, adapters and connections that implement one or more *ConnectorServices*. All *ConnectableObjects* implement at least the *TimeConnectorService*.
ActiveObject	The base class that serves as the super class for all PowerMatcher objects with active behaviour that implements the *SchedulerConnectorService*. An active object uses the scheduler to schedule one-time or periodic tasks, like for example a PowerMatcher agent that schedules a periodic task to send its next bid update. Objects with strictly reactive behaviour, so only reacting to external events, extend *ConnectableObject* directly or indirectly.
IdentifiableObject	The base class that serves as the super class for all identifiable PowerMatcher components like agents, adapters and connections. *IdentifiableObject* defines common configuration properties like cluster and component ID, and utility methods for logging.

5.1.4 The supply-demand matching sequence

The general supply-demand matching interaction sequence for the PowerMatcher agent is explained in Figure 16 and Figure 17, differentiating between the start-up sequence and the normal sequence that follows it.

The lifecycle of an agent is generally the following.

1. The agent is instantiated. It is not yet configured and it is passive.
2. The agent is configured[1], but it remains passive.

[1] In the OSGi runtime environment, when the configuration of an agent is changes it is destroyed and recreated automatically by default. Technically it is possible to handle configuration changes without recreating the agent, but no example has been provided for this.

3. A matcher (adapter) binds to the agent. The agent becomes active and goes through the start-up sequence of the PowerMatcher protocol.
4. The agent has received its initial market basis and executes its normal supply-demand matching sequence.
5. The matcher (adapter) unbinds from the agent. The agent stops its supply-demand matching sequence.
6. The agent instance is deleted.

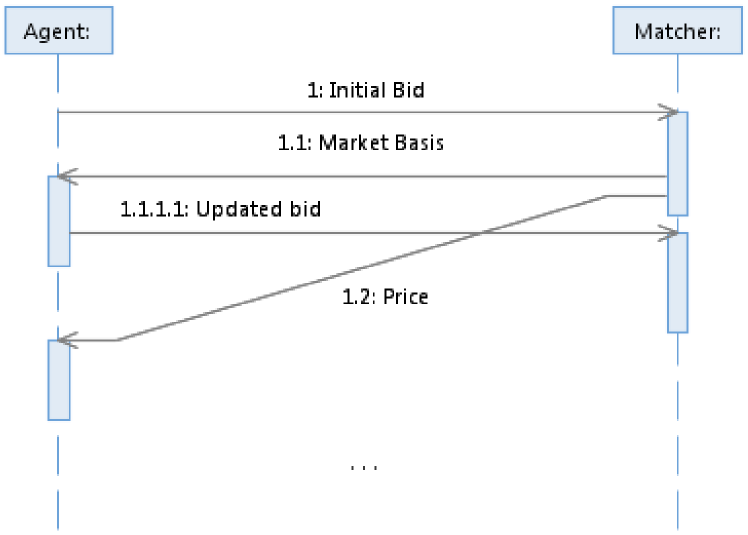


Figure 16: Agent supply-demand matching start-up sequence

In the start-up sequence, the agent needs to wait for the initial market basis before it can send out its initial bid. The parent matcher publishes the current market basis on a regular interval, so the agent could simply wait for the next update. However, as shown in Figure 16, the agent can optionally send an initial empty bid to the parent matcher to trigger the parent matcher to publish the current market basis (and price immediately).

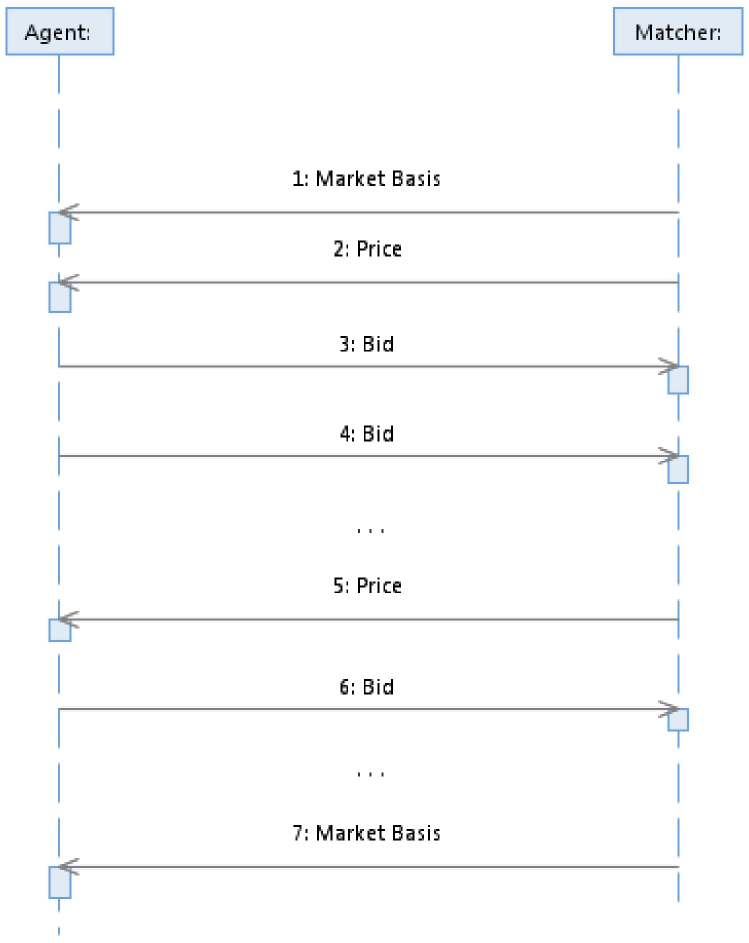


Figure 17: Agent supply-demand matching follow-up sequence

Once the initial market basis has been received, the agent can publish an updated bid at any time, for example, when conditions of the DER the agent controls change. If there are no changes in the environment, the agent must repeat its last bid at some minimum interval. The default implementation in the *Agent* framework class is to re-publish the last bid at the configured update interval (see *ActiveObject*).

No specific action is required[1] upon deactivation of the agent. The parent matcher discards the last bid and forgets about the agent if it has not received a bid update within a cleanup interval.

5.2 Writing the Agent

To accelerate agent development, template Java projects are made available that can be used as the starting point for developing a new Agent and any custom Adapters it may require. This section introduces the development of a new agent based on these templates. The PowerMatcher Core Installation and Configuration manual[3] provides instructions for setting up the prerequisite Eclipse development

[1] This may change in the 1.0 release of the reference implementation.

environment, as well as detailed information about configuring and running the new agent in a PowerMatcher cluster.

5.2.1 Agent and Adapter template projects

The following table lists the template projects that are provided in PowerMatcher Core and PowerMatcher Extension to develop new Agents and Adapters.

Template project	Description
net.powermatcher.core.agent.template	Provides two core example agents, one without a custom adapter and one with a custom adapter.
net.powermatcher.core.agent.template.component	Exports the core example agents as components in the OSGi Service Component Model.
net.powermatcher.core.messaging.adapter.template	Provides an example adapter and its factory that uses publish-subscribe messaging to communicate with the environment
net.powermatcher.core.messaging.adapter.template.component	Exports the example adapter factory as a component in the OSGi Service Component Model.
net.powermatcher.agent.template	Provides an example of an agent that uses the PowerMatcher Extension Telemetry interface to report measurement and status values.
net.powermatcher.agent.template.component	Exports the extension example agent as component in the OSGi Service Component Model.

As the table illustrates, an Agent and the Adapter are always developed in (at least) two separate projects:

1. A project implementing the logic of the Agent or the Adapter as Plain Old Java Objects (POJOs), depending only on the Java Standard Edition profile, and
2. A project wrapping the POJO Agent or Adapter as a configurable component in the OSGi Service Component Model for Declarative Services[4].

Running PowerMatcher Agents as POJOs supports static configurations and simple testing and debugging. Running PowerMatcher Agents as OSGi components supports dynamic configuration and provisioning of PowerMatcher solutions in a distributed environment.

5.2.2 Steps to create a new agent or adapter from the template projects

In general, a new agent or adapter is developed in the following steps:

1. Select the template agent or adapter to be used as the starting point and copy the *template* and *template.component* projects to projects with a new name.
2. Rename the packages and agent or adapter class name.
3. Implement the new agent or adapter by modifying the agent or adapter classes and their configuration interfaces.

A brief introduction to the available template agent and adapter classes is given in the following sections.

5.2.3 ExampleAgent1 – an elementary agent

The *net.powermatcher.core.agent.template* project is a template project that implements 2 example agents.

Interface	Description
ExampleAgent1	An agent that publishes a step-shaped bid for a configurable demand and price value.
ExampleAgent2	An extension of *ExampleAgent1* that implements a control and notification interface for integration of this agent with an adapter.

Class *ExampleAgent1*, is described here and shown in the Java listing below.

ExampleAgent1 publishes a step-shaped bid curve that is defined by two parameters, *bidPrice* and *bidPower*.

- If *bidPower* is a positive value, it defines how many Watts the resource will consume if the market price is less than *bidPrice*.
- If *bidPower* is a negative value, it defines how many Watts the resource will supply if the market price is equal or higher than than *bidPrice*.

The bid curve that *ExampleAgent1* publishes is shown in Figure 18, depending on the sign of the configured value for *bidPower*.

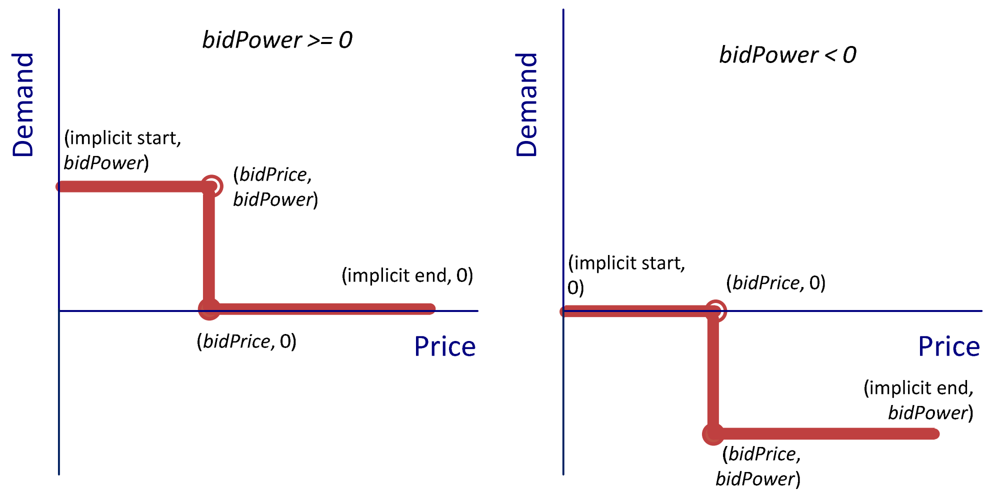


Figure 18: Step-shaped bid of ExampleAgent1

In the listing, the fields declared on line 6 and 10 store the configured values for *bidPrice* and *bidPower*. When the agent is configured, the framework calls the *setConfiguration* method defined on line 94. This in turn calls the *initialize* method on line 79, which is the method that initialized the *bidPrice* and *bid-Power* fields with either the configured value, or the default value if not explicitly specified in the configuration.

Listing 1: ExampleAgent1 class of the net.powermatcher.core.agent.template project

```
1   public class ExampleAgent1 extends Agent {
2
3   /**
4   The bid price is a configuration property for the agent.
5   */
6   private double bidPrice;
7   /**
8   The bid power is a configuration property for the agent.
9   */
10  private double bidPower;
11
12  /**
13  Constructs an unconfigured instance of this class.
14  */
15  public ExampleAgent1() {
16  super();
17  }
18
19  /**
20  Do update. This method is intended for updating the agents status
21  and publish a new bid reflecting that status.
22  */
23  @Override
24  protected void doUpdate() {
25  /*
26  The super method should always be called to record the last
27  update time.
28  */
29  super.doUpdate();
30
31  /*
Get the current market basis, which is the last market basis
received by this agent.
*/
MarketBasis marketBasis = getCurrentMarketBasis();
if (marketBasis != null) {

/*
Convert the configured bid price to normalized price units (npu).
Normalize price unit 0 is the price step that corresponds to price 0.0.
*/
int normalizedPrice = marketBasis.toNormalizedPrice(
this.bidPrice);
normalizedPrice = marketBasis.boundNormalizedPrice(
normalizedPrice);

/*
Construct a step-shaped bid that consist of 2 price points.
*/
PricePoint pricePoint1;
PricePoint pricePoint2;

if (this.bidPower >= 0) {
pricePoint1 = new PricePoint(normalizedPrice,
this.bidPower);
pricePoint2 = new PricePoint(normalizedPrice, 0);
} else {
pricePoint1 = new PricePoint(normalizedPrice, 0);
pricePoint2 = new PricePoint(normalizedPrice,
this.bidPower);
}

BidInfo newBidInfo = new BidInfo(marketBasis, pricePoint1,
pricePoint2);

/*
Publish the new bid to the matcher.
If no matcher is connected to the agent, the bid is silently discarded.
*/
publishBidUpdate(newBidInfo);
logInfo("Published new bid " + newBidInfo);
}
}

/**
Initialize the agent by setting the agent's fields with the
configured properties or their default values, if applicable.
*/
private void initialize() {
/*
Get the configured bid step price, or the default value if not configured.
*/
this.bidPrice = getProperty(ExampleAgent1Configuration.
                      BID_PRICE_PROPERTY,
                      ExampleAgent1Configuration.BID_PRICE_DEFAULT);
/*
Get the configured bid step demand, or the default value if not configured.
*/
this.bidPower = getProperty(ExampleAgent1Configuration.
                      BID_POWER_PROPERTY,
                      ExampleAgent1Configuration.BID_POWER_DEFAULT);
}

/**
Configures the agent.
*/
```

```
93  @Override
94  public void setConfiguration(final ConfigurationService configuration) {
95  super.setConfiguration(configuration);
96  initialize();
97  }
98
99  /**
100 Update market basis with the specified new market basis parameter.
101 This callback method is invoked when the agent receives its initial
102 or updated market basis.<br>
103 This implementation extends the default behavior, which is to
104 set the new market basis, by logging the updated market basis.
105 */
106 @Override
107 public void updateMarketBasis(final MarketBasis newMarketBasis) {
108 logInfo("New market basis received:" + newMarketBasis);
109 super.updateMarketBasis(newMarketBasis);
110 }
111
112 /**
113 Update price info with the specified new price info parameter.
114 This callback method is invoked when the agent receives its initial
115 or updated market price.<br>
116 */
117 @Override
118 public void updatePriceInfo(final PriceInfo newPriceInfo) {
119 logInfo("New price info received:" + newPriceInfo);
120 super.updatePriceInfo(newPriceInfo);
121 }
122
```

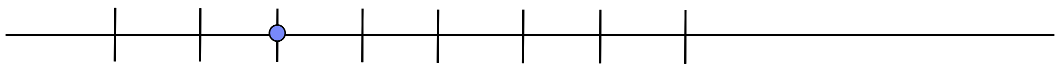
Once the agent becomes active, the framework invokes the *doUpdate* method at the configured update interval (property *update.interval*, default is 30 seconds). On line 35-36, the agent checks if it has already received a market basis from its parent matcher.

The *updateMarketBasis* and *updatePriceInfo* methods on line 107 and 118 are called by the framework when the agent receives an update from its parent matcher via the agent interface. In ExampleAgent1 these methods simply invoke their super methods, which store the new market basis and price in field variables. Normally one would control the resource from the *updatePriceInfo* to react to changes in price. This is covered in *ExampleAgent2*.

Once the initial market basis has been received, the agent constructs and publishes its bid curve in line 37-71. A price is usually expressed as a floating point value, but can also be expressed using a step index relative to the market basis. The price can be expressed as index in two alternative ways.

1. As price step, starting at index 0 for the minimum price.
2. As normalized price unit (NPU), with an index of 0 for the price closest to 0.0

This is illustrated in the example of Figure 19.



Price	€ - 0.20	€ - 0.10	€ 0.00	€ 0.10	€ 0.20	€ 0.30	€ 0.40	€ 0.50
Step	0	1	2	3	4	5	6	7
Npu	-2	-1	0	1	2	3	4	5

Figure 19: Price, price step and NPU example

The bid curve that is constructed on line 53-64 is defined by the points in the bid curve of Figure 18. The points are represented using NPUs with *PricePoint*s. On line 42-45, utility methods of *MarketBasis* support the conversion of the configured *bidPrice* to NPU, and bounding of the configured price within the minimum and maximum of the price range defined by the market basis.

Line 70 publishes the bid via the matcher interface to the parent matcher.

5.2.4 ExampleAgent2 – an agent using a custom adapter

The *net.powermatcher.core.agent.template* project is a template project that implements 2 example agents.

Interface	Description
ExampleAgent1	An agent that publishes a step-shaped bid for a configurable demand and price value.
ExampleAgent2	An extension of *ExampleAgent1* that implements a control and notification interface for integration of this agent with an adapter.

Class *ExampleAgent2*, is described here and shown in the Java listing below.

ExampleAgent2 defines an interface for an adapter, as implemented through the following interfaces.

Interface	Description
ExampleConnectorService	Defines a connector to interface with an external distributed energy resource. In this example the connector interface lets the adapter register a notification listener with the agent, and get a control interface to interact with the agent. On line 28 and 47 *ExampleAgent2* implements the bind and unbind methods that are called by the adapter to bind the notification listener to the agent.
ExampleNotificationService	Defines an example method *somethingChanged* that is called by the agent when the market price is updated. The adapter's notification listener implements this method.
ExampleControlService	Defines an example method *doSomething* that the adapter can call to interact with the agent. This method is implemented by *ExampleAgent2* on line 9.

Listing 2: ExampleAgent2 class of the net.powermatcher.core.agent.template project

```
1   public class ExampleAgent2 extends ExampleAgent1 implements ExampleConnectorService {
2
3           private class ExampleServiceImpl implements ExampleControlService {
```

```java

                /* (non-Javadoc)
                 * @see net.powermatcher.core.agent.template.service.ExampleControlService#doSomething()
                 */
                @Override
                public void doSomething() {
                        logInfo("Adapter called doSomething");
                }

        }

        /**
         * Define the example adapter (ExampleNotificationService) field.
         * This field references the adapter that is currently bound to the agent, or
         * is null if no adapter is bound.
         */
        private ExampleNotificationService exampleAdapter;

        /**
         * Bind an adapter's notification interface to the agent.
         * This implementation supports only one adapter being bound at the same time.
         * @see net.powermatcher.core.agent.template.service.ExampleConnectorService#bind(net.powermatcher.core.agent.template.service.ExampleNotificationService)
         */
        @Override
        public void bind(final ExampleNotificationService exampleAdapter) {
                assert this.exampleAdapter == null;
                this.exampleAdapter = exampleAdapter;
        }

        /**
         * Gets the agent's control interface (ExampleControlService).
         * @see net.powermatcher.core.agent.template.service.ExampleConnectorService#getExampleService()
         */
        @Override
        public ExampleControlService getExampleService() {
                return new ExampleServiceImpl();
        }

        /**
         * Unbind an adapter's notification interface from the agent.
         * @see net.powermatcher.core.agent.template.service.ExampleConnectorService#unbind(net.powermatcher.core.agent.template.service.ExampleNotificationService)
         */
        @Override
        public void unbind(final ExampleNotificationService exampleAdapter) {
                this.exampleAdapter = null;
        }

        /**
         * Update price info with the specified new price info parameter.
         * This implementation extends the super behavior by notifying the adapter that something has changed.
         * @see net.powermatcher.core.agent.template.ExampleAgent1#updatePriceInfo(net.powermatcher.core.agent.framework.data.PriceInfo)
         */
        @Override
        public void updatePriceInfo(final PriceInfo newPriceInfo) {
                ExampleNotificationService exampleAdapter = this.exampleAdapter;
                if (exampleAdapter != null) {
                        logInfo("Notifying adapter that something has changed");
                        exampleAdapter.somethingChanged();
                }
                super.updatePriceInfo(newPriceInfo);
        }

}
```

The following projects provide a template implementation of a custom adapter and its adapter factory for the *ExampleConnectorService* that is exported by the example agent.

1. *net.powermatcher.core.messaging.adapter.template*
2. *net.powermatcher.core.messaging.adapter.template.component*

These template projects serve as the starting point for developing a new adapter, for example for interfacing to a device. The example adapter uses a messaging connector to interface with the outside world, and sends a message to itself. The template adapter and its adapter factory is not covered in this manual; please refer to the documentation embedded in the source code of the template projects for further information.

5.2.5 ExampleAgent3 – an agent using the extension Telemetry adapter

The *net.powermatcher.agent.template* project is a template project that defines *ExampleAgent3*, an extension of the core *ExampleAgent1* that also publishes an update count as telemetry measurement data. This example is included of the PowerMatcher Extension library, as it depends on the Telemetry feature, which is a feature that is provided as-is. Previous examples are part of the reference implementation, and are therefore provided in the PowerMatcher Core library.

ExampleAgent3 uses the Telemetry service interface to publish measurement and status data. The following interfaces are defined by the Telemetry service.

Interface/Class	Description
TelemetryConnectorService	Defines a connector for publishing telemetry events. On line 30 and 80 *ExampleAgent3* implements the bind and unbind methods that are called by the adapter to bind the telemetry listener of the adapter to the agent.
TelemetryListenerConnectorService	Defines a connector for receiving telemetry events. This interface is not implemented by *ExampleAgent3*, but is, for example, implemented by the *net.powermatcher.agent.telemetry.logging* agent that logs telemetry events to comma-separated value files.
TelemetryService	Defines a single method *processTelemetryData* for publishing telemetry data via the telemetry connector or receiving via the telemetry listener connector.
TelemetryDataPublisher	A class providing utility methods to publish different types of telemetry events to the telemetry service. *ExampleAgent3* wraps the

	telemetry service interface in a publisher in the *bind* method, on line 31. On line 69 and 64 the *ExampleAgent3* uses the publisher to report a status event (a key-string value pair) and a measurement event (a key-float value pair) each time the *doUpdate* method is called.

Listing 3: ExampleAgent3 class of the net.powermatcher.agent.template extension project

```
1   public class ExampleAgent3 extends ExampleAgent1 implements TelemetryConnectorService {
2
3          /**
4           * The update count that is published is a dimensionless measurement, and
    therefore the
5           * unit that is reported is empty.
6           */
7          private static final String NO_UNIT = "";
8
9          /**
10          * The telemetry data publisher publishes the telemetry events to the telemetry
    adapter.
11          * The publisher is a utility class that wraps the telemetry service interface
    that
12          * is provided by the adapter that connects to this agent.
13          */
14         private TelemetryDataPublisher telemetryDataPublisher;
15
16         /**
17          * The update count is incremented for each update event and published as
    telemetry measurement value.
18          */
19         private int updateCount;
20
21         /**
22          * Bind the specified telemetry publisher.
23          *
24          * @param telemetryPublisher
25          *            The telemetry publisher (<code>TelemetryService</code>)
26          *            to bind.
27          * @see #unbind(TelemetryService)
28          */
29         @Override
30         public void bind(TelemetryService telemetryPublisher) {
31                this.telemetryDataPublisher = new
    TelemetryDataPublisher(getConfiguration(), telemetryPublisher);
32         }
33
34         /**
35          * Do update. This method is intended for updating the agents status
36          * and publish a new bid reflecting that status. It is periodically invoked
37          * by the framework at the configured update interval.
38          *
39          * This method extends the behavior in the superclass by publishing and
    incrementing the
40          * updat count.
41          *
42          * @see net.powermatcher.core.agent.template.ExampleAgent1#doUpdate()
43          */
44         @Override
```

```
    protected void doUpdate() {
        TelemetryDataPublisher telemetryDataPublisher = this.telemetryDataPublisher;

/*
 * If a telemetry adapter has connected to the agent, publish a status event for
 * the update and publish the update count as a measurement.
 */
        if (telemetryDataPublisher != null) {
            logInfo("Publishing status update " + this.updateCount);
            Date now = new Date();
/*
 * Publish a status event.
 * A status event has a valueName (the key), a string value and a timestamp.
 */
            telemetryDataPublisher.publishStatusData("action", "updating", now);
/*
 * Publish a measurement event.
 * A measurement event has a valueName (the key), a numeric value, a unit and a timestamp.
 */
            telemetryDataPublisher.publishMeasurementData("updateCount", NO_UNIT, Float.valueOf(this.updateCount), null, now);
        }
        this.updateCount += 1;
        super.doUpdate();
    }

    /**
     * Unbind the specified telemetry publisher.
     *
     * @param telemetryPublisher
     *            The telemetry publisher (<code>TelemetryService</code>)
     *            to unbind.
     *
     * @see #bind(TelemetryService)
     */
    @Override
    public void unbind(TelemetryService telemetryPublisher) {
        this.telemetryDataPublisher = null;
    }

```

5.2.6 Agent and Adapter configuration

The configuration interface defines the information for the agent or adapter that cannot be obtained from the device directly, or configuration information that is not related to the agent, such as tuning parameters for the control logic.

Interface	Description
ExampleAgent1Configuration	Defines keys and default values of the configuration properties for *ExampleAgent1*. It is shown in the listing below. This interface extends *AgentConfiguration*, and thereby inherits the configuration properties that are common to all agents, including the cluster and agent id.

ExampleAgent2Configuration	Defines keys and default values of the configuration properties for *ExampleAgent2*. As *ExampleAgent2* does not introduce any new properties, this interface is empty.
AgentConfiguration	Defines the configuration properties that are common to all agents, including the cluster and agent id (inherited from *BaseObjectConfiguration* via *ActiveObjectConfiguration*).
AdapterConfiguration	Defines the configuration properties that are common to all adapter, including the cluster and adapter id (inherited from *IdentifiableObjectConfiguration*).
MessagingAdapterConfiguration	Defines the configuration properties that are common to all adapter, including the cluster and adapter id (inherited from *IdentifiableObjectConfiguration*).

The configuration properties of the core and extension agents and adapters are described in the PowerMatcher Core Installation and Configuration manual[3].

For each property the configuration interfaces defines

- The key of the configuration property, as shown for example on line 5.
- For optional properties, the default value, as shown for example on line 9.
- The signature of an access method for the property, as shown for example on line 35. It is not necessary to implement the access method anywhere in the code, but the access method is used to automatically generate the required meta type information for OSGi components. This requires that the method name is the same as the key for the configuration property, where any '.' In the key must be replaced by '_'.

Listing 4: Configuration interface for ExampleAgent1 in the net.powermatcher.core.agent.template project

```
1   public interface ExampleAgent1Configuration extends AgentConfiguration {
2          /**
3           * Define the bid price property (String) constant.
4           */
5          public static final String BID_PRICE_PROPERTY = "bid.price";
6          /**
7           * Define the bid price default (double) constant.
8           */
9          public static final double BID_PRICE_DEFAULT = 0.50;
10         /**
11          * Define the bid price default (String) constant.
12          * The default value is also defined as a string literal to allow the default
13          * value to be referenced in OSGi configuration meta type annotations.
14          */
15         public static final String BID_PRICE_DEFAULT_STR = "0.50";
16         /**
17          * Define the bid power property (String) constant.
18          */
19         public static final String BID_POWER_PROPERTY = "bid.power";
20         /**
21          * Define the bid power default (double) constant.
```

```
         */
        public static final double BID_POWER_DEFAULT = 100.0;
        /**
         * Define the bid power default str (String) constant.
         */
        public static final String BID_POWER_DEFAULT_STR = "100";

        /**
         * Access method for the <code>bid.price</code> property.
         * This method is not implemented anywhere, but provides the signature for OSGi
metatype annotations.
         *
         * @return The value configured for the <code>bid.price</code> property.
         */
        public double bid_price();

        /**
         * Access method for the <code>bid.power</code> property.
         * This method is not implemented anywhere, but provides the signature for OSGi
metatype annotations.
         *
         * @return The value configured for the <code>bid.power</code> property.
         */
        public double bid_power();

}
```

5.2.7 Component definition for the OSGi Service Component Model

The OSGi Declarative Service model[4] supports the life cycle of agents and adapters by creating, activating, configuring and destroying components as configurations are created, updated and deleted from the OSGi Configuration Admin service. The OSGi Configuration Admin service distinguishes between two different types of components.

- OSGi Managed Service components. These are singleton components that are optionally configured. An example of a singleton component is the PowerMatcher Configuration Management agent, of which there is only one per PowerMatcher runtime.
- OSGi Factory Service components. These are components from which multiple instances are created as configurations are added to the configuration registry. Examples of this are PowerMatcher Device Agents or Adapter Factories. Multiple instances can be created for a particular type of device agent, as more devices are added. Multiple instances can be created for a particular type of adapter factory, in case multiple factories are needed (for example one protocol adapter factory for each different PowerMatcher cluster deployed on a single PowerMatcher runtime). So, OSGi Factory Service components are implemented to create adapter factories: a factory of factories.

The OSGi configuration registry is updated by a separate configuration agent, and can be inspected and manipulated with, for example, the Apache Felix Web Console.

To enable an agent or adapter for deployment in the OSGi environment, a component and a configuration definition must be provided. Below the definitions are described and listed to enable *ExampleAgent1* as OSGi component.

Interface	Description
ExampleAgent1Component	Defines *ExampleAgent1* as OSGi factory service component using aQute Bnd annotations that are supported by Eclipse Bndtools. When the bundle is built, the OSGI-INF scm xml is generated to define the component and its activation and deactivation method on line 19 and 30.
ExampleAgent1ComponentConfiguration	The Object Class Definition (OCD) that defines the configuration properties for the component. When the bundle is built, the OSGI-INF/metatype xml is generated to define the configuration properties and defaults associated with the configuration identifier of the component (defined by COMPONENT_NAME). The metatype information allows the configuration of the component to be defined and inspected via management applications, like for example the Apache Felix Web Console.

Listing 5: Component class for ExampleAgent1 in the net.powermatcher.core.agent.template.component project

```
1   @Component(name = ExampleAgent1Component.COMPONENT_NAME, designateFactory =
    ExampleAgent1ComponentConfiguration.class)
2   public class ExampleAgent1Component extends ExampleAgent1 {
3           /**
4            * Define the component name (String) constant.
5            * The fully qualified name of the PowerMatcher agent is used, as it is also the
    basis for the
6            * persistent identifier in the OSGi configuration repository.
7            */
8           public final static String COMPONENT_NAME =
    "net.powermatcher.core.agent.example.ExampleAgent1";
9
10          /**
11           * Activate an instance of the agent with the specified configuration properties.
12           * This method is called from OSGi whenever a configuration item is created or
    updated in the
13           * OSGi configuration repository.
14           *
15           * @param properties
16           *            The configuration properties (<code>Map<String,Object></code>) for
    the agent.
17           */
18          @Activate
19          void activate(final Map<String, Object> properties) {
20                  ConfigurationService configuration = new BaseConfiguration(properties);
21                  setConfiguration(configuration);
22          }
23
24          /**
25           * Deactivate an instance of the agent.
26           * This method is called from OSGi whenever a configuration item is deleted or
    about
27           * to be updated in the OSGi configuration repository.
28           */
29          @Deactivate
```

```
	void deactivate() {
		/* do nothing */
	}

}
```

Listing 6: Component configuration for ExampleAgent1 in the net.powermatcher.core.agent.template.component project

```
@OCD(name = "PowerMatcher Example Agent 1")
public interface ExampleAgent1ComponentConfiguration extends ExampleAgent1Configuration {

	@Override
	@Meta.AD(required = false, deflt = CLUSTER_ID_DEFAULT, description = 
CLUSTER_ID_DESCRIPTION)
	public String cluster_id();

	@Override
	@Meta.AD(required = true, description = ID_DESCRIPTION)
	public String id();

	@Override
	@Meta.AD(required = false, deflt = ENABLED_DEFAULT_STR, description = 
ENABLED_DESCRIPTION)
	public boolean enabled();

	@Override
	@Meta.AD(required = false, deflt = BID_PRICE_DEFAULT_STR, description = 
BID_PRICE_DESCRIPTION)
	public double bid_price();

	@Override
	@Meta.AD(required = false, deflt = BID_POWER_DEFAULT_STR, description = 
BID_POWER_DESCRIPTION)
	public double bid_power();

	@Override
	@Meta.AD(required = false, deflt = UPDATE_INTERVAL_DEFAULT_STR, description = 
UPDATE_INTERVAL_DESCRIPTION)
	public int update_interval();

	@Override
	@Meta.AD(required = false, deflt = PARENT_MATCHER_ID_DEFAULT_STR, description = 
PARENT_MATCHER_ID_DESCRIPTION)
	public String matcher_id();

	@Override
	@Meta.AD(required = false, deflt = TIME_ADAPTER_FACTORY_DEFAULT, description = 
TIME_ADAPTER_FACTORY_DESCRIPTION)
	public String time_adapter_factory();

	@Override
	@Meta.AD(required = false, deflt = SCHEDULER_ADAPTER_FACTORY_DEFAULT, description 
= SCHEDULER_ADAPTER_FACTORY_DESCRIPTION)
	public String scheduler_adapter_factory();

	@Override
	@Meta.AD(required = false, deflt = AGENT_ADAPTER_FACTORY_DEFAULT, description = 
AGENT_ADAPTER_FACTORY_DESCRIPTION)
	public String agent_adapter_factory();

	@Override
	@Meta.AD(required = false, deflt = LOGGING_ADAPTER_FACTORY_DEFAULT, description = 
LOGGING_ADAPTER_FACTORY_DESCRIPTION)
```

```
        public String logging_adapter_factory();

        @Override
        @Meta.AD(required = false, deflt = LOG_LISTENER_ID_DEFAULT_STR, description =
    LOG_LISTENER_ID_DESCRIPTION)
        public String log_listener_id();

        @Override
        @Meta.AD(required = false, deflt = AGENT_BID_LOG_LEVEL_DEFAULT, description =
    AGENT_BID_LOG_LEVEL_DESCRIPTION,
                        optionValues = { NO_LOGGING, PARTIAL_LOGGING, FULL_LOGGING },
    optionLabels = {
                        NO_LOGGING_LABEL,
                        PARTIAL_LOGGING_LABEL,
                        FULL_LOGGING_LABEL })
        public String agent_bid_log_level();

        @Override
        @Meta.AD(required = false, deflt = AGENT_PRICE_LOG_LEVEL_DEFAULT, description =
    AGENT_PRICE_LOG_LEVEL_DESCRIPTION, optionValues = { NO_LOGGING, FULL_LOGGING },
    optionLabels = {
                        NO_LOGGING_LABEL,
                        FULL_LOGGING_LABEL })
        public String agent_price_log_level();

}
```

The template adapter factory component and component configuration are not covered in this manual. The component class for an adapter factory contains some additional logic (inherited from the adapter framework) to track connectors in the OSGi Service Registry, and bind to and unbind from the connector or connectors that it serves. Please refer to the documentation embedded in the source code of the template projects for further information.

5.3 Running the Agent

Template projects are provided in PowerMatcher Core and PowerMatcher Extension to develop new runtime configurations for Agents and Adapters, and to test these runtime configurations from the Eclipse workspace. These template projects provide an example configuration that creates a PowerMatcher cluster containing *ExampleAgent1*, *ExampleAgent2*, *ExampleAgent3*, *Concentrator*, *ObjectiveAgent*, *Auctioneer*, *CSVLoggingAgent* and *TelemetryCSVLoggingAgent*. Figure 20 shows the PowerMatcher agents and logical relationships that are created in the Extension configuration. The Core configuration is the same as the Extension configuration, minus ExampleAgent3 and the TelemetryCSVLoggingAgent.

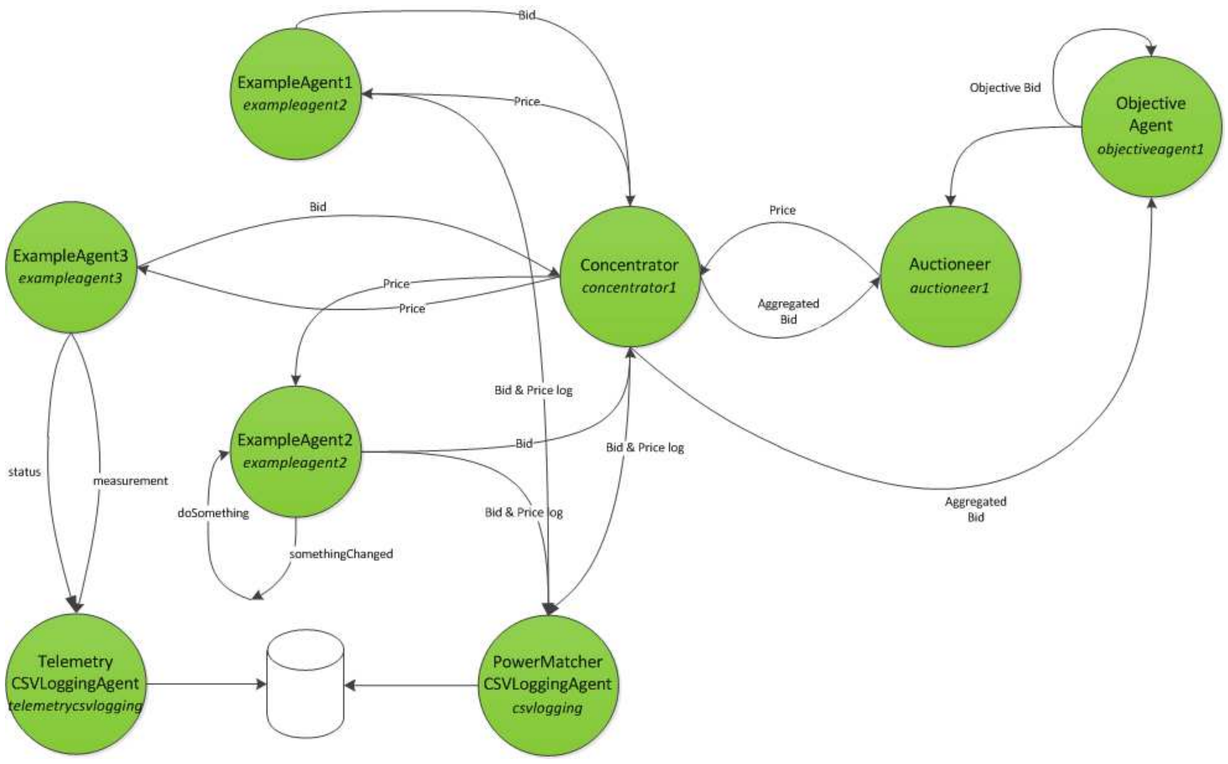


Figure 20: The example PowerMatcher runtime configuration provided in the template projects

5.3.1 Template projects for running Agents and Adapters

Two different runtime environments are supported to run a PowerMatcher application.

1. A Java SE runtime environment. This environment uses a Java main Launcher class to create and configure a PowerMatcher cluster of agents and adapters. A single configuration properties file defines the agents to be instantiated, with the configuration properties of the agent and all its adapters. The Launcher class implements the logic to instantiate adapters, depending on the connectors implemented by the agents.

   This is a simple environment with a static configuration, for example suitable for testing and debugging, or for unmanaged deployments.

2. An OSGi runtime environment. This environment uses management agents to dynamically install software and configuration updates, in combination with other means to inspect and define configurations, like for example the Apache Felix Web Console. Example projects are provided that launch an OSGi runtime environment from a bundle list, and instantiate a PowerMatcher configuration from a configuration xml file in the project.

Please refer to the PowerMatcher Core Installation and Configuration manual[3] for a detailed description of the configuration properties and schema, and for launching the PowerMatcher application in different ways.

The following table lists the template projects that are provided in PowerMatcher Core and Power-Matcher Extension to develop new runtime configurations for Agents and Adapters, and to test these runtime configurations from the Eclipse workspace. These projects can be used as-is to run the template agents and adapters, or can be used as the starting point for new development by copying and renaming the desired project.

Template project	Description
net.powermatcher.core.launcher.main.template	Java SE runtime configuration and launcher for running core example agents and adapter against an external MQ Telemetry Transport message broker.
net.powermatcher.core.launcher.osgi.template	OSGi runtime configuration and launcher for running core example agents and adapter against an external MQ Telemetry Transport message broker.
net.powermatcher.extension.launcher.main.template	Java SE runtime configuration and launcher for running core and extension example agents and adapter against an external MQ Telemetry Transport message broker.
net.powermatcher.extension.launcher.osgi.template	OSGi runtime configuration and launcher for running core example agents and adapters against an external MQ Telemetry Transport message broker.
net.powermatcher.expeditor.launcher.mbroker.template	Java SE runtime configuration and launcher for running core and extension example agents and adapters with an embedded IBM MicroBroker MQTT message broker.
net.powermatcher.expeditor.launcher.osgi.template	OSGi runtime configuration and launcher for running core and extension example agents and adapters with an embedded IBM Micro-Broker MQTT message broker.

5.3.2 PowerMatcher application and configuration on the Java SE runtime

The three Java SE launcher template projects contain an Eclipse launcher that launches a Java class with a *main* method to start the PowerMatcher runtime, and a configuration file *agent_config.properties* that the launcher class uses to instantiate and configure a PowerMatcher cluster. The three Java SE launcher template projects (see 5.3.2 for the description of the projects) are the following:

1. *net.powermatcher.core.launcher.main.template*
2. *net.powermatcher.extension.launcher.main.template*
3. *net.powermatcher.expeditor.launcher.mbroker.template*

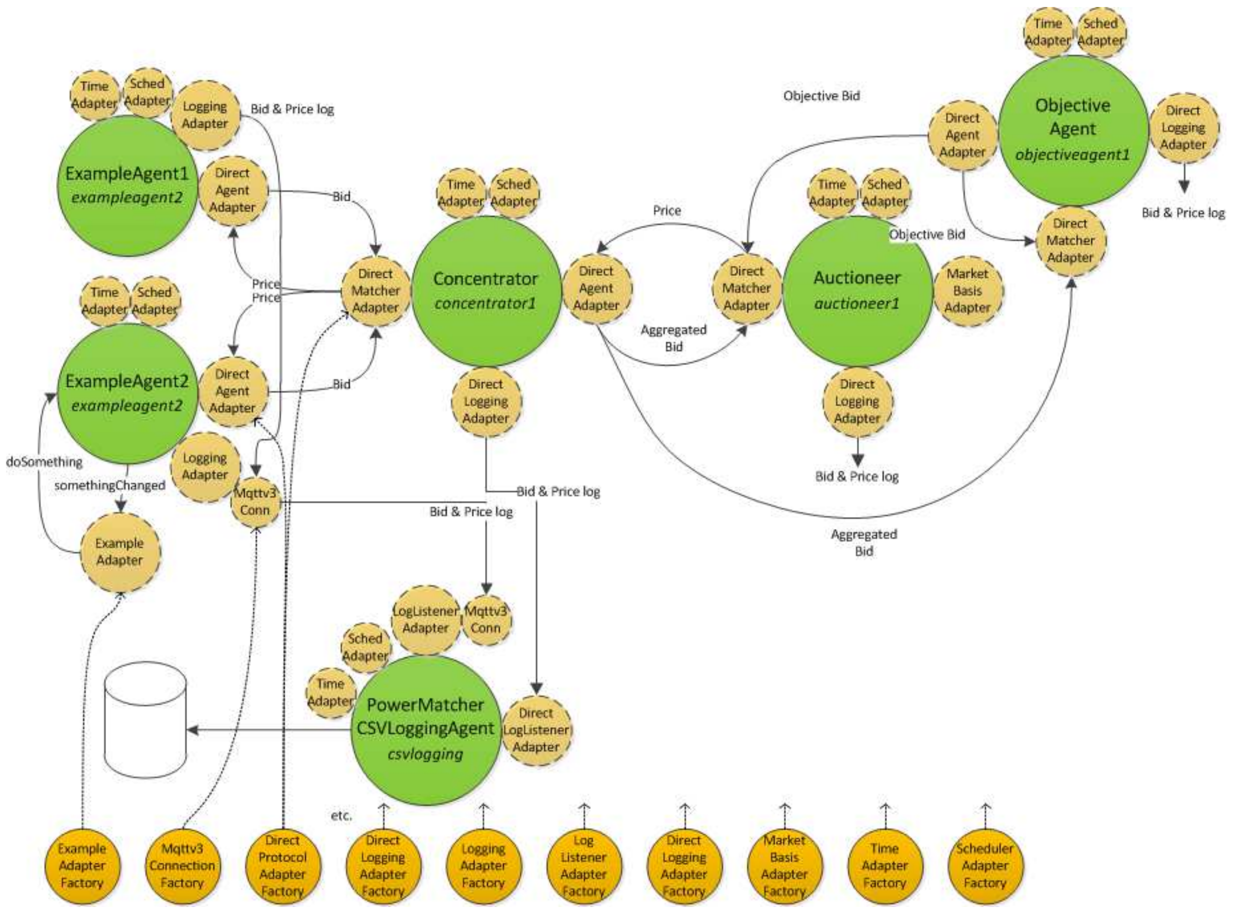


Figure 21: The *net.powermatcher.core.launcher.main.template* configuration

The *agent_config.properties* file, of which an example is shown in listing 7, contain definitions that instantiate the agents shown in green in Figure 21 for the *net.powermatcher.core.launcher.main.template* project, and in Figure 22 for the *net.powermatcher.extension.launcher.main.template* and *net.powermatcher.expeditor.launcher.mbroker.template* projects. In Figure 22, the configuration elements that are the same as in Figure 21 have been greyed out.

The adapters, shown in these diagrams in light yellow with a dashed border, are automatically instantiated and connected to the agents by the framework from the configured adapter factories. The adapter factories are shown in dark yellow.

The Java SE configuration is static, that is, once the PowerMatcher runtime has been launched the configuration cannot be changed anymore.

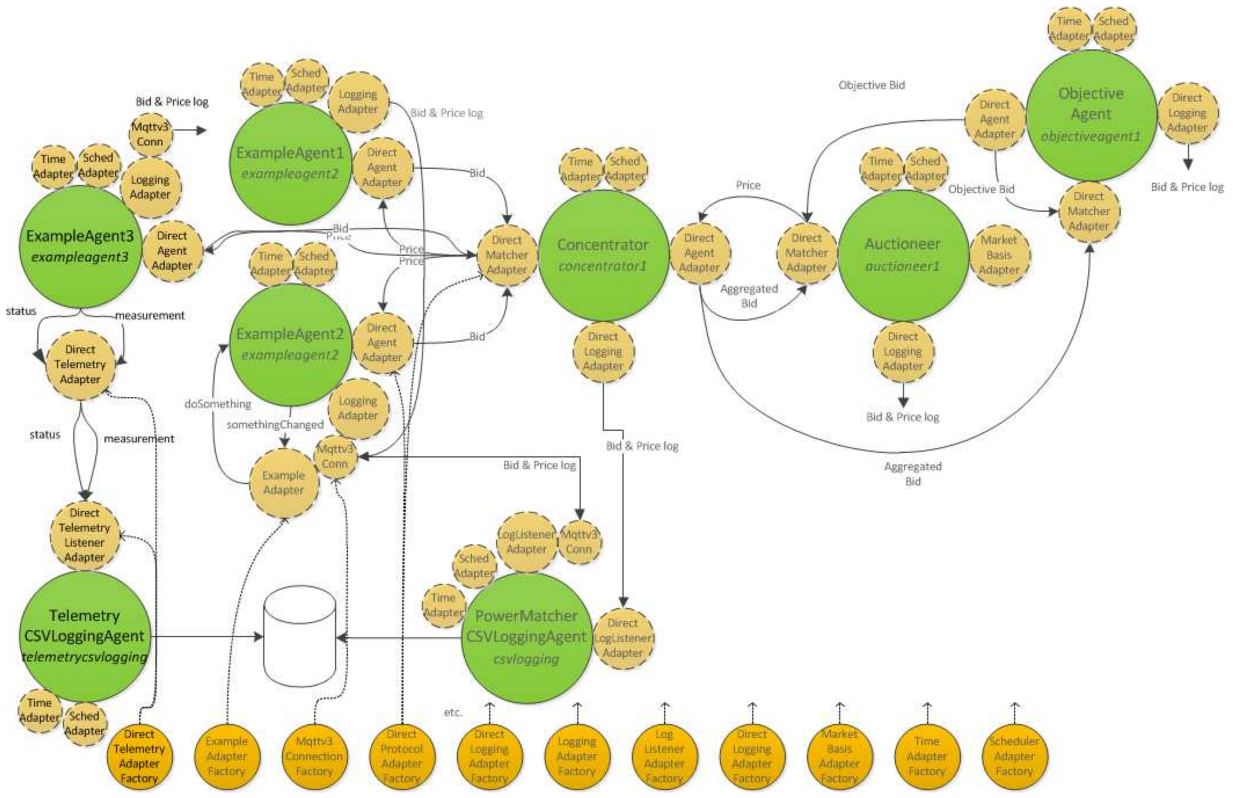


Figure 22: The *net.powermatcher.extension.launcher.main.template* configuration

Listing 7: Core example agent_configuration.properties for the Java SE Launcher.

```
# Define time & scheduler adapter factories
adapter.factory.time.class=net.powermatcher.core.scheduler.TimeAdapterFactory
adapter.factory.scheduler.class=net.powermatcher.core.scheduler.SchedulerAdapterFactory

# Define adapter factories for direct connections, without using message broker
adapter.factory.marketbasis.class=net.powermatcher.core.agent.marketbasis.adapter.MarketBasisAdapterFactory
adapter.factory.directprotocol.class=net.powermatcher.core.direct.protocol.adapter.DirectProtocolAdapterFactory
adapter.factory.directlogging.class=net.powermatcher.core.direct.protocol.adapter.DirectLoggingAdapterFactory

# Define adapter factories for message broker connections
adapter.factory.agentmsgprotocol.class=net.powermatcher.core.messaging.protocol.adapter.AgentProtocolAdapterFactory
adapter.factory.agentmsgprotocol.agent.messaging.protocol=INTERNAL_v1
adapter.factory.matchermsgprotocol.class=net.powermatcher.core.messaging.protocol.adapter.MatcherProtocolAdapterFactory
adapter.factory.matchermsgprotocol.matcher.messaging.protocol=INTERNAL_v1
adapter.factory.loggingmsgprotocol.class=net.powermatcher.core.messaging.protocol.adapter.LoggingAdapterFactory
adapter.factory.loggingmsgprotocol.connector.id=loggingAdapter
adapter.factory.loglistenermsgprotocol.class=net.powermatcher.core.messaging.protocol.adapter.LogListenerAdapterFactory
adapter.factory.mqttv3connection.class=net.powermatcher.core.messaging.mqttv3.Mqttv3ConnectionFactory

# Define project-specific adapter factories
adapter.factory.objectiveadapter.class=
adapter.factory.peakshavingadapter.class=
adapter.factory.exampleadapter.class=net.powermatcher.core.messaging.adapter.template.ExampleAdapterFactory

# Unless specific values are configured for individual agents, the default values in the comments below are used.
# Example of agent-specific factory configuration can be found for auctioneer1 and concentrator1.

# Default scheduler adapter factories
time.adapter.factory=time
scheduler.adapter.factory=scheduler

# Default adapter factories for direct connections, without using message broker
agent.adapter.factory=directprotocol
logging.adapter.factory=directlogging

# Default adapter factories for message broker connections
#agent.adapter.factory=agentmsgprotocol
#matcher.adapter.factory=matchermsgprotocol
#logging.adapter.factory=loggingmsgprotocol
log.listener.adapter.factory=loglistenermsgprotocol
messaging.adapter.factory=mqttv3connection

# Default project-specific adapter factories
#objective.adapter.factory=objectiveadapter
#peak.shaving.adapter.factory=peakshavingadapter
example.adapter.factory=exampleadapter


# Global properties.
cluster.id=ExampleCluster
# Change from default 300 seconds to 10 seconds
update.interval=10
# By default logging will go to the configured CSVLoggingAgent agent.
log.listener.id=csvlogging

# CSVLoggingAgent configuration
agent.csvlogging.class=net.powermatcher.core.agent.logging.CSVLoggingAgent
#agent.csvlogging.cluster.id=
agent.csvlogging.id=csvlogging
agent.csvlogging.enabled=true
#agent.csvlogging.update.interval=300
agent.csvlogging.powermatcher.bid.logging.pattern='pwm_bid_log_'yyyyMMdd'.csv'
agent.csvlogging.powermatcher.price.logging.pattern='pwm_price_log_'yyyyMMdd'.csv'
#agent.csvlogging.list.separator=;
#agent.csvlogging.date.format=yyyy-MM-dd HH:mm:ss

# Auctioneer1 configuration
agent.auctioneer1.class=net.powermatcher.core.agent.auctioneer.Auctioneer
agent.auctioneer1.id=auctioneer1
agent.auctioneer1.enabled=true
#agent.auctioneer1.cluster.id=
#agent.auctioneer1.matcher.id=root
agent.auctioneer1.agent.adapter.factory=marketbasis
# Price in currency, to define the market basis
agent.auctioneer1.commodity=electricity
agent.auctioneer1.currency=EUR
agent.auctioneer1.minimum.price=0.00
agent.auctioneer1.maximum.price=0.99
agent.auctioneer1.price.steps=100
agent.auctioneer1.significance=2
agent.auctioneer1.market.ref=0
#agent.auctioneer1.matcher.agent.bid.log.level=FULL_LOGGING
agent.auctioneer1.matcher.aggregated.bid.log.level=FULL_LOGGING
#agent.auctioneer1.matcher.price.log.level=FULL_LOGGING
```

```

# Concentrator1 configuration
agent.concentrator1.class=net.powermatcher.core.agent.concentrator.Concentrator
agent.concentrator1.id=concentrator1
agent.concentrator1.enabled=true
# The concentrator has 2 direct agent adapters, one to the auctioneer, one to the objective agent.
agent.concentrator1.agent.adapter.factory=directprotocol,directprotocol
agent.concentrator1.matcher.id=auctioneer1,objectiveagent1
# Changed from 30 seconds to 10 for test in workspace
#agent.concentrator1.update.interval=30
agent.concentrator1.update.interval=10
#agent.concentrator1.agent.bid.log.level=NO_LOGGING
#agent.concentrator1.agent.price.log.level=NO_LOGGING
#agent.concentrator1.matcher.agent.bid.log.level=FULL_LOGGING
#agent.concentrator1.matcher.aggregated.bid.log.level=NO_LOGGING
#agent.concentrator1.matcher.price.log.level=FULL_LOGGING

# ObjectiveAgent configuration
agent.objectiveagent1.class=net.powermatcher.core.agent.objective.ObjectiveAgent
agent.objectiveagent1.id=objectiveagent1
agent.objectiveagent1.enabled=true
agent.objectiveagent1.matcher.id=auctioneer1
# Prices in normalized price units
#agent.objectiveagent1.objective.bid=(price1,demand1);(price2,demand2);(price3,demand3);...
agent.objectiveagent1.objective.bid=(0,-50.0)
#agent.objectiveagent1.agent.bid.log.level=NO_LOGGING
#agent.objectiveagent1.agent.price.log.level=NO_LOGGING
agent.objectiveagent1.matcher.agent.bid.log.level=NO_LOGGING
#agent.objectiveagent1.matcher.aggregated.bid.log.level=NO_LOGGING
agent.objectiveagent1.matcher.price.log.level=NO_LOGGING

# Example Agent 1 configuration
agent.exampleagent1.class=net.powermatcher.core.agent.template.ExampleAgent1
agent.exampleagent1.id=exampleagent1
agent.exampleagent1.enabled=true
agent.exampleagent1.logging.adapter.factory=loggingmsgprotocol
agent.exampleagent1.matcher.id=concentrator1
#agent.exampleagent1.bid.price=0.50
#agent.exampleagent1.bid.power=100
#agent.exampleagent1.agent.bid.log.level=NO_LOGGING
#agent.exampleagent1.agent.price.log.level=NO_LOGGING

# Example Agent 2 configuration
agent.exampleagent2.class=net.powermatcher.core.agent.template.ExampleAgent2
agent.exampleagent2.id=exampleagent2
agent.exampleagent2.enabled=true
agent.exampleagent2.logging.adapter.factory=loggingmsgprotocol
agent.exampleagent2.matcher.id=concentrator1
#agent.exampleagent2.bid.price=0.50
#agent.exampleagent2.bid.power=100
#agent.exampleagent2.agent.bid.log.level=NO_LOGGING
#agent.exampleagent2.agent.price.log.level=NO_LOGGING
agent.exampleagent2.example.setting=ExampleValue
agent.exampleagent2.example.setting=ExampleValue
```

5.3.3 PowerMatcher application and configuration on the OSGi runtime

The OSGi launcher template projects contain a Bndtools *bndrun* launcher that launches an OSGi runtime containing all of the bundles that are required to run the configuration.

The three OSGi launcher template projects (see 5.3.2 for the description of the projects) that are provided are the following:

1. *net.powermatcher.core.launcher.osgi.template*
2. *net.powermatcher.extension.launcher.osgi.template*
3. *net.powermatcher.expeditor.launcher.osgi.template*

The configuration management agent in the *net.powermatcher.core.config.management.agent* bundle is configured to process an XML configuration specified via a URL. The *bndrun* launcher specifies a file URL that points to the *conf-local/example_core_config.xml* file in the project.

The configuration management agent uses the configuration XML, from which a small excerpt is shown in listing 9, to instantiate and configure a PowerMatcher cluster.

Listing 8: Excerpt from the example_core_config.xml for the OSGi launcher.

```
<?xml version="1.0" encoding="UTF-8"?>
<nodeconfig id="example-node" name="Example node"
        description="PowerMatcher example node" date="2012-08-29 16:02:06">
. . .
        <!-- ExampleCluster Core Example Agents Group -->

        <configuration type="group" cluster="ExampleCluster"
                id="core_example_agents">
                <configuration type="factory" cluster="ExampleCluster"
                        pid="net.powermatcher.core.agent.example.ExampleAgent1" id="exampleagent1">
                        <property name="id" value="exampleagent1" type="String"></property>
                        <property name="bid.power" value="100" type="Double"></property>
                        <property name="bid.price" value="0.50" type="Double"></property>
                </configuration>
                <configuration type="factory" cluster="ExampleCluster"
                        pid="net.powermatcher.core.agent.example.ExampleAgent2" id="exampleagent2">
                        <property name="id" value="exampleagent2" type="String"></property>
                        <property name="bid.price" value="0.50" type="Double"></property>
                        <property name="bid.power" value="100" type="Double"></property>
                        <property name="example.setting" value="Example!" type="String"></property>
        </configuration>
. . .
</nodeconfig>
```

The OSGi configuration is dynamic, that is, the configuration management agent dynamically reconfigures the PowerMatcher cluster when the configuration XML is updated. Furthermore, there is no need for a *configureAgent* method to creates and connect adapters as in the Java SE launcher. The creations and configuration of agents and adapters, as well as any other OSGi components, are fully specified in the configuration XML. New and updated agent and adapter bundles can be dynamically loaded or unloaded in the OSGi runtime, and they can be dynamically added or removed from the configuration XML.

The OSGi launcher template projects create the same configurations as shown in Figure 21 and Figure 22, but in the configuration XML the configuration is represented by a tree of configuration groups and configuration elements. The diagram of Figure 23 shows the XML configuration hierarchy of the *net.powermatcher.core.launcher.osgi.template* project, and the diagram of Figure 24 the hierarchy in the *net.powermatcher.extension.launcher.osgi.template* and *net.powermatcher.expeditor.launcher.osgi .template* projects. In Figure 24, the configuration elements that are the same as in Figure 23 have been greyed out.

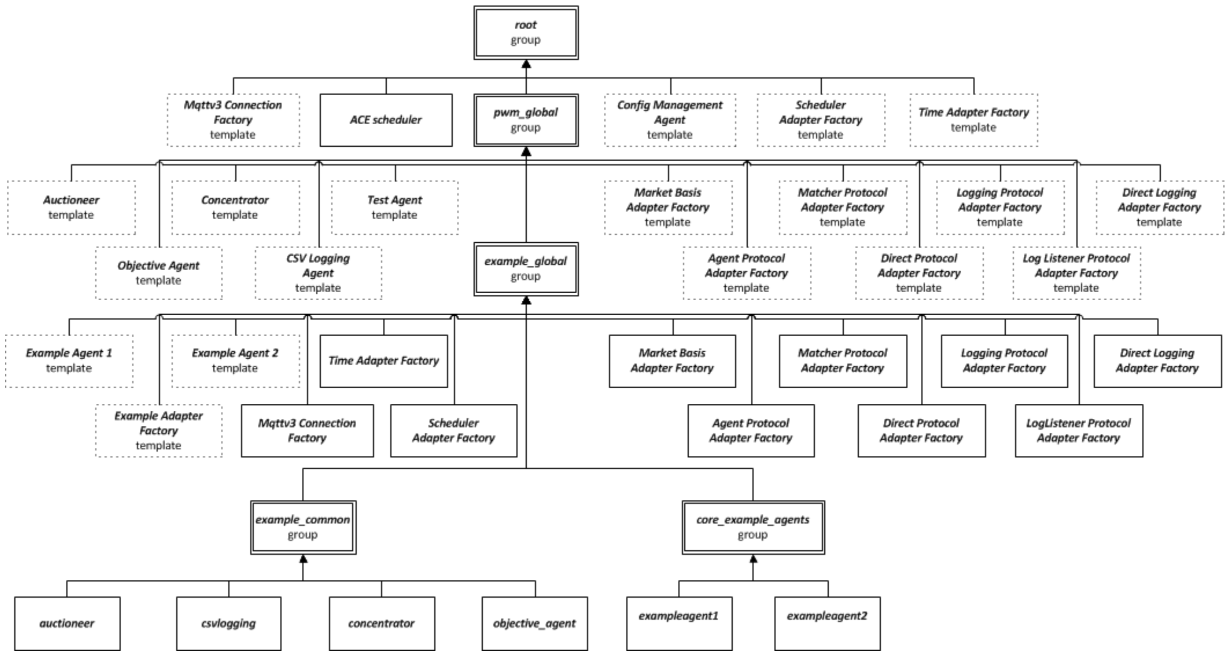

root group
Mqttv3 Connection Factory template
ACE scheduler
pwm_global group
Config Management Agent template
Scheduler Adapter Factory template
Time Adapter Factory template
Auctioneer template
Concentrator template
Test Agent template
Market Basis Adapter Factory template
Matcher Protocol Adapter Factory template
Logging Protocol Adapter Factory template
Direct Logging Adapter Factory template
Objective Agent template
CSV Logging Agent template
example_global group
Agent Protocol Adapter Factory template
Direct Protocol Adapter Factory template
Log Listener Protocol Adapter Factory template
Example Agent 1 template
Example Agent 2 template
Time Adapter Factory
Market Basis Adapter Factory
Matcher Protocol Adapter Factory
Logging Protocol Adapter Factory
Direct Logging Adapter Factory
Example Adapter Factory template
Mqttv3 Connection Factory
Scheduler Adapter Factory
Agent Protocol Adapter Factory
Direct Protocol Adapter Factory
LogListener Protocol Adapter Factory
example_common group
core_example_agents group
auctioneer
csvlogging
concentrator
objective_agent
exampleagent1
exampleagent2

Figure 23: The *net.powermatcher.core.launcher.osgi.template* XML configuration hierarchy

The configuration XML defines different types of configuration elements:

- A configuration group defines a scope for configuration elements below the group. A configuration group can define default values for properties for the scope of the group. An example of this is the definition of *cluster.id=ExampleCluster* in configuration group ***example_global***, which effectively assigns all agents and adapters in this group to ExampleCluster.
- A template configuration defines the default values for the configuration properties of a specific type of component, designed by persistent identifier (pid), for all occurrences below the group. An example of this is property *update.interval=30* in ***auctioneer_template*** and other templates, which defines the default update interval of 30 seconds for all auctioneers in all clusters, because the template is defined at the ***pwm_global*** level.
- A configuration defines the instantiation and configuration of a component of a specific type, designed by persistent identifier (pid), like for example a specific agent or an adapter factory. All simple rectanges in diagram represent configurations. Here all the properties are defined that are not already defined by template or global property default higher in the hierarchy.

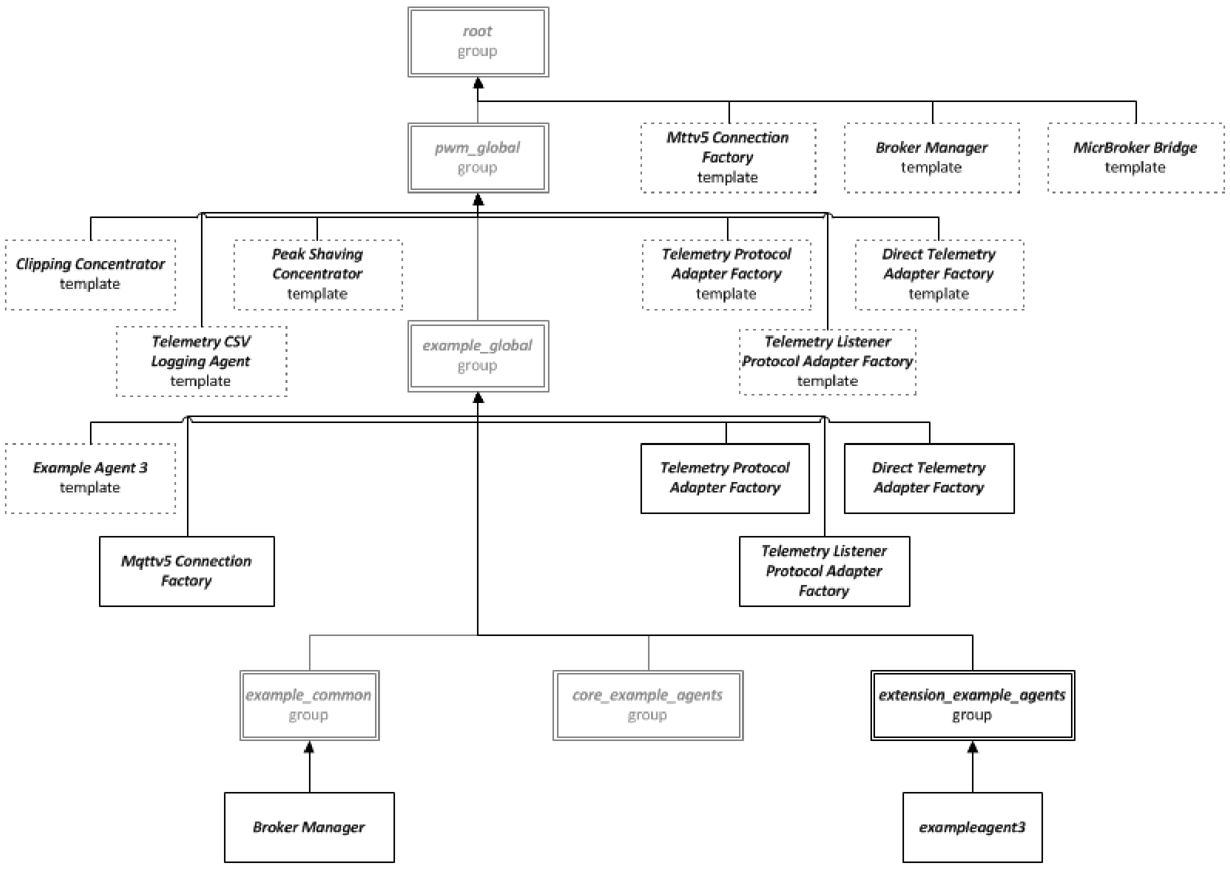


Figure 24: The *net.powermatcher.extension.launcher.osgi.template* XML configuration hierarchy

6 Further reading

Theoretical background

Koen Kok, Martin Scheepers, and René Kamphuis. *Intelligence in electricity networks for embedding renewables and distributed generation.* Chapter in: R.R. Negenborn, Z. Lukszo, and J. Hellendoorn, editors, Intelligent Infrastructures. Springer, Intelligent Systems, Control and Automation: Science and Engineering Series, 2009. (download)

Koen Kok, Cor Warmer, and René Kamphuis. The PowerMatcher: *Multi-agent control of electricity demand and supply.* IEEE Intelligent Systems, 21(2):89–90, Part of overview article: "Agents in Industry: The Best from the AAMAS 2005 Industry Track''. (download)

PowerMatcher field demonstration results

Cor Warmer, Maarten Hommelberg, Bart Roossien, Koen Kok and Jan-Willem Turkstra, *A field test using agents for coordination of residential micro-chp*, Proceedings of the International Conference on Intelligent System Applications to Power Systems (ISAP), Kaohsiung, Taiwan, 4-8 November 2007. (download)

Bart Roossien, Pamela Macdougall, Albert van den Noort, Frits Bliek and Rene Kamphuis, *Intelligent Heating Systems in Households for Smart Grid Applications*, Proceedings of I-SUP, Bruges, Belgium 18-21 April, 2010. (download)

Koen Kok, Zsofia Derzsi, Jaap Gordijn, Maarten Hommelberg, Cor Warmer, René Kamphuis, and Hans Akkermans, *Agent-based electricity balancing with distributed energy resources, a multiperspective case study,* in Ralph H. Sprague, editor, Proceedings of the 41st Annual Hawaii International Conference on System Sciences, page 173, Los Alamitos, CA, USA, 2008. IEEE Computer Society. (download)

References

[1]	PowerMatcher - Communication Protocol Specification – r0.7v5; IBM & TNO.	24-10-2012
[2]	MQ Telemetry Transport (MQTT) V3.1 Protocol Specification; IBM. See http://mqtt.org/	19-08-2010
[3]	PowerMatcher Core - Installation and Configuration r0.7v1.doc; IBM.	15-10-2012
[4]	OSGi Service Platform, Service Compendium, Release 4, Version 4.2; The OSGi Alli-	08-2009

ance