Comparative Analysis of Coretime Price Simulator and the Broker Pallet Implementation

Introduction

This comprehensive report provides an in-depth comparative analysis of two distinct implementations: the `broker pallet` code in Rust and the Coretime price simulator in Python. The primary objective of this comparison is to elucidate the similarities and differences between these two implementations, with a particular focus on their functionality. The Python-based price simulator is specifically designed to mimic the broker pallet code, offering insights into potential future Polkadot Fellowship PR requests for code improvements.

Documentation Approach

Each function in the Python price-simulation that corresponds to a function in the broker pallet is meticulously documented. This documentation includes detailed comments within the function definitions and links to the analogous function in the Rust implementation of the broker pallet.

Comparative Analysis

1. *poly.py vs. adapt_price.rs*

   Linear Functionality:
   Both implementations have a `leadin_factor_at` function that calculates the lead-in factor.
   The adapt_price function in both cases adjusts the price based on the number of cores sold.

   Exponential Functionality:
   Only implemented in Python.

   Summary:
   The Python implementation in poly.py closely mirrors the functionality of the Rust implementation in the adapt_price.rs of the broker pallet. The Python implementation includes an additional example of exponential functionality and factor, which is not present in the Rust implementation, but is there to

showcase the different potential scenarios of how different implementations would work and are there to serve

2. *price.py*

2.1 Function `update_renewal_price` vs `do_renew` (in dispatchable_impls.rs in broker pallet)

Similarities:
Both implementations calculate a price_cap as a result of a price calculation with a renewal bump factor.
Both implementations use the min function to set the new renewal price, considering the calculated price_cap.

Differences:
The Python code is meant to imitate part of the code in the broker pallet:

```
let price_cap = record.price + config.renewal_bump * record.price;
let price = Self::sale_price(&sale, now).min(price_cap);
```

It uses self.initial_bought_price and self.new_buy_price in its calculations, reflecting the internal state of the Python class.
The Rust code references record.price and involves a call to Self::sale_price(&sale, now) as part of the price calculation.

Summary:
The Python code in price.py closely mirrors the essential functionality found in the Rust code in dispatchable_impls.rs. Both implementations calculate a capped renewal price based on certain factors. The Python code directly uses class attributes, while the Rust code involves interactions with other functions specific to the broader context of the broker pallet.

2.2 Function rotate_sale vs rotate_sale (in tick_impls.rs in broker pallet)

The python function only imitates part of the functionality in the broker pallet that relates to changes in pricing. Both implementations serve the purpose of calculating the starting price for the upcoming sale based on the number of cores sold. Both check conditions related to the offered cores, sold cores, and ideal cores.

Similarities:
Both implementations handle scenarios where no cores are offered for sale or where cores are sold beyond the ideal amount.
Both implementations involve calculations based on the sold cores, ideal cores, and offered cores to determine the purchase price.

Differences:

The Python code directly uses class attributes (self.config, self.cores_sold, etc.), reflecting the internal state of the Python class.
The Rust code involves interactions with other functions specific to the broader context of the broker pallet and relies on generic types (T::PriceAdapter).

2.3 Function __sale_price_calculate vs do_purchase and sale_price (in dispatchable_impls.rs and utility_impls.rs in broker pallet)

Similarities:
Both implementations calculate the variable num as the difference between the current block time and the sale start time, clamped to the lead-in length.
Both calculate the through parameter based on the calculated num and lead-in length.
Both involve the calculation of the sale price using the lead-in factor (LF) and the current price.

Differences:
The Python code directly uses class attributes (self.config, self.linear, self.factor, self.price), reflecting the internal state of the Python class.
The Rust code uses generic types (T) and Rust-specific types (BlockNumberFor<T>, BalanceOf<T>) due to the broader context of the broker pallet.

Summary:
The Python code in __sale_price_calculation closely mirrors the essential functionality found in the Rust code in utility_impls.rs. Both implementations calculate the sale price at a given block time based on similar calculations of the lead-in factor (LF).

On the other hand, since the function in python calls the __sellout_price_update function, it imitates part of the do_purchase function in the broker pallet.

Similarities:
Both implementations check if the condition for setting the sellout price is met.
Both involve setting the sellout price based on certain conditions.

Differences:
The Python code directly uses class attributes (self.config, self.cores_sold_in_renewal, self.cores_sold_in_sale, self.sellout_price, self.price), reflecting the internal state of the Python class.
The Rust code interacts with the sale struct, which contains specific fields (cores_sold, ideal_cores_sold, sellout_price, etc.) relevant to the context of the broker pallet.

Summary:

The Python code in __sellout_price_update closely mirrors the essential functionality found in the Rust code within the do_purchase function. Both implementations aim to update the sellout price under specific conditions.

2.4 Function __renew_price vs do_renew (in dispatchable_impls.rs in broker pallet)

Similarities:
Both implementations calculate the new buy price using a similar formula: `price_cap = record.price + config.renewal_bump * record.price.`
Both involve retrieving the current block number (in Rust using frame_system::Pallet::<T>::block_number() and in Python as a parameter block_now).

Differences:
The Rust code directly interacts with specific types and traits from the Substrate framework (frame_system::Pallet::<T>::block_number()).
The Python code utilizes class attributes (self.initial_bought_price, self.config.renewal_bump, self.new_buy_price, self.__sale_price_calculate) and internal state.

Summary:
The Python code in __renew_price closely mirrors the essential functionality found in the Rust code within the do_renew function. Both implementations calculate the new buy price after renewal using a similar formula, and the differences lie in the specific elements they interact with (class attributes vs. Substrate framework types).

Conclusion

This report highlights that the Python and Rust implementations share fundamental similarities in functionality. However, they differ in aspects such as state management, interaction with external functions, and specific implementation contexts. The Python simulator effectively mimics the broker pallet's functionality, providing valuable insights for future enhancements and Polkadot Fellowship PR requests.