Data Platform Actors

Zhifeng Zhang

Oct. 13, 2024

Contents

Data Platform Actors

Producer Actor

The **Producer Actor** serves as the backbone of data ingestion in the platform. Its primary role is to acquire both historical and real-time tick data from external sources, specifically **CryptoCompare**. This actor employs two methods of communication to gather data: **REST APIs** for historical data and **WebSockets** for real-time streams.

- **States:**
  - **Idle:** The producer awaits instructions or triggers to begin data fetching or topic creation.
  - **Fetching Data:** Upon receiving a command, the producer initiates the retrieval of data from CryptoCompare.
  - **Publishing to Fixed Topics:** After successfully fetching data, the producer publishes this information to predefined Redpanda topics designated for fixed data streams.
  - **Publishing to Dynamic Topics:** In scenarios requiring dynamic data streams, the producer routes data to dynamically created Redpanda topics based on runtime needs.
  - **Creating Dynamic Topic:** Handles requests to establish new topics dynamically, allowing the system to adapt to varying data requirements.
  - **Error:** Captures and manages any issues that arise during data fetching or publishing processes.
- **Events:**
  - **FETCH_DATA:** Triggers the initiation of data retrieval.
  - **CREATE_DYNAMIC_TOPIC:** Requests the creation of a new dynamic topic.
  - **DATA_READY:** Indicates successful data retrieval.
  - **PUBLISH_SUCCESS:** Confirms that data has been successfully published to a topic.
  - **PUBLISH_FAILURE:** Signals a failure in publishing data.
  - **CREATE_TOPIC_SUCCESS:** Confirms the successful creation of a dynamic topic.

  - **CREATE_TOPIC_FAILURE:** Indicates a failure in creating a dynamic topic.
  - **RESET:** Resets the producer from an error state back to idle.

- **Transitions:**
  - **Idle → Fetching Data:** Triggered by the `FETCH_DATA` event.
  - **Idle → Creating Dynamic Topic:** Triggered by the `CREATE_DYNAMIC_TOPIC` event.
  - **Fetching Data → Publishing to Fixed Topics:** Upon successful data retrieval (`DATA_READY`).
  - **Fetching Data → Error:** If data fetching fails (`FETCH_FAILURE`).
  - **Publishing to Fixed Topics → Idle:** Upon successful publishing (`PUBLISH_SUCCESS`).
  - **Publishing to Fixed Topics → Error:** If publishing fails (`PUBLISH_FAILURE`).
  - **Publishing to Dynamic Topics → Idle:** Upon successful publishing (`PUBLISH_SUCCESS`).
  - **Publishing to Dynamic Topics → Error:** If publishing fails (`PUBLISH_FAILURE`).
  - **Creating Dynamic Topic → Idle:** Upon successful topic creation (`CREATE_TOPIC_SUCCESS`).
  - **Creating Dynamic Topic → Error:** If topic creation fails (`CREATE_TOPIC_FAILURE`).
  - **Error → Idle:** Resetting after an error (`RESET`).

The **Producer Actor** ensures continuous data flow by handling both scheduled data fetches and on-demand topic creations. Its ability to adapt to dynamic requirements makes the data ingestion process flexible and resilient.

```
\documentclass{standalone}
\usepackage[svgnames]{xcolor} % Enables a wide range of color names
\usepackage{tikz}
\usetikzlibrary{positioning, shapes.geometric, arrows, automata}

\begin{document}

\begin{tikzpicture}[>=stealth, node distance=3cm, auto, scale=0.8, transform shape]

  % Define styles
  \tikzstyle{state} = [
    rectangle,
    rounded corners,
    minimum width=2.5cm,
    minimum height=1cm,
    draw,
    fill=LightBlue,
    font=\tiny
  ]
  \tikzstyle{trans} = [
    ->,
    >=stealth,
    thick,
    font=\tiny
  ]

  % Define states
  \node[state] (Idle) {Idle};
  \node[state, right=3cm of Idle] (FetchingData) {Fetching Data};
  \node[state, below right=3cm and 1cm of FetchingData] (PublishingFixed) {Publishing to Fixed Topics};
  \node[state, above right=3cm and 1cm of FetchingData] (PublishingDynamic) {Publishing to Dynamic Topics};
  \node[state, right=3cm of PublishingDynamic] (CreatingTopic) {Creating Dynamic Topic};
  \node[state, below of=PublishingFixed, yshift=-1cm, color=Red, fill=Red!20] (Error) {Error};

  % Define transitions
  \path[->, trans]
    (Idle) edge node {FETCH\_DATA} (FetchingData)
    (Idle) edge node {CREATE\_DYNAMIC\_TOPIC} (CreatingTopic)
```

```
    (FetchingData) edge node {DATA\_READY} (PublishingFixed)
    (FetchingData) edge [bend left=45] node {FETCH\_FAILURE} (Error)

    (PublishingFixed) edge node {PUBLISH\_SUCCESS} (Idle)
    (PublishingFixed) edge node {PUBLISH\_FAILURE} (Error)

    (PublishingDynamic) edge node {PUBLISH\_SUCCESS} (Idle)
    (PublishingDynamic) edge node {PUBLISH\_FAILURE} (Error)

    (CreatingTopic) edge node {CREATE\_TOPIC\_SUCCESS} (Idle)
    (CreatingTopic) edge node {CREATE\_TOPIC\_FAILURE} (Error)

    (Error) edge node {RESET} (Idle);

\end{tikzpicture}

\end{document}
```

Coordinator Actor

The **Coordinator Actor** orchestrates the overall data flow and manages dynamic aspects of the system, ensuring seamless interactions between producers and consumers.

- **States:**
  - **Active:** The coordinator is operational and ready to handle incoming requests for dynamic topic creation.
  - **Processing Create Topic:** Upon receiving a request, the coordinator begins the process of creating a new dynamic topic.
  - **Notifying Actors:** After successfully creating a topic, the coordinator informs both the **Producer Actor** and relevant **Consumer Actors** to utilize the new topic.
  - **Error:** Manages any issues encountered during the topic creation or notification processes.
- **Events:**
  - **REQUEST_CREATE_TOPIC:** Initiates the creation of a new dynamic topic based on application needs.
  - **CREATE_TOPIC_SUCCESS:** Confirms that a dynamic topic has been successfully created.
  - **CREATE_TOPIC_FAILURE:** Indicates a failure in creating a dynamic topic.
  - **NOTIFY_SUCCESS:** Confirms that actors have been successfully notified about the new topic.
  - **NOTIFY_FAILURE:** Signals a failure in notifying actors about the new topic.
  - **RESET:** Resets the coordinator from an error state back to active.
- **Transitions:**
  - **Active Processing Create Topic:** Triggered by the `REQUEST_CREATE_TOPIC` event.
  - **Processing Create Topic Notifying Actors:** Upon successful topic creation (`CREATE_TOPIC_SUCCESS`).
  - **Processing Create Topic Error:** If topic creation fails (`CREATE_TOPIC_FAILURE`).
  - **Notifying Actors Active:** Upon successful notifications (`NOTIFY_SUCCESS`).
  - **Notifying Actors Error:** If notifications fail (`NOTIFY_FAILURE`).
  - **Error Active:** Resetting after an error (`RESET`).

By managing dynamic topic creation and ensuring that all relevant actors are informed and synchronized, the **Coordinator Actor** plays a crucial role in maintaining the flexibility and scalability of the data platform.

```
\documentclass{standalone}
\usepackage[svgnames]{xcolor} % Enables a wide range of color names
\usepackage{tikz}
\usetikzlibrary{positioning, shapes.geometric, arrows, automata}

\begin{document}
```

```
\begin{tikzpicture}[>=stealth, node distance=4cm, auto, scale=0.8, transform shape]

  % Define styles
  \tikzstyle{state} = [
    rectangle,
    rounded corners,
    minimum width=2.8cm,
    minimum height=1cm,
    draw,
    fill=LightGreen,
    font=\tiny
  ]
  \tikzstyle{trans} = [
    ->,
    >=stealth,
    thick,
    font=\tiny
  ]

  % Define states
  \node[state] (Active) {Active};
  \node[state, right=4cm of Active] (ProcessingCreateTopic) {Processing Create Topic};
  \node[state, below right=4cm and 1cm of ProcessingCreateTopic] (NotifyingActors) {Notifying Actors};
  \node[state, right=4cm of ProcessingCreateTopic, yshift=-1cm, color=Red, fill=Red!20] (Error) {Error};

  % Define transitions
  \path[->, trans]
    (Active) edge node {REQUEST\_CREATE\_TOPIC} (ProcessingCreateTopic)
    (ProcessingCreateTopic) edge node {CREATE\_TOPIC\_SUCCESS} (NotifyingActors)
    (ProcessingCreateTopic) edge node {CREATE\_TOPIC\_FAILURE} (Error)

    (NotifyingActors) edge node {NOTIFY\_SUCCESS} (Active)
    (NotifyingActors) edge node {NOTIFY\_FAILURE} (Error)

    (Error) edge node {RESET} (Active);

\end{tikzpicture}

\end{document}
```

Consumer for Fixed Topics Actor

This actor is responsible for handling data from fixed, predefined topics. It ensures that tick data is accurately aggregated and stored efficiently for future access and analysis.

- **States:**
  - **Idle:** The consumer awaits incoming messages from fixed topics.
  - **Processing Message:** Upon receiving a message, the consumer processes the tick data.
  - **Writing to DB:** After processing, the aggregated data is written to **TimescaleDB**.
  - **Updating Redis:** Concurrently, the consumer updates **Redis** with the latest aggregated data to facilitate quick retrieval.
  - **Error:** Manages any issues that arise during message processing, database writing, or cache updating.
- **Events:**
  - **MESSAGE_RECEIVED:** Indicates that a new tick message has been received from a fixed topic.

  - **PROCESS_SUCCESS:** Confirms successful processing of the tick message.
  - **PROCESS_FAILURE:** Signals a failure in processing the tick message.
  - **WRITE_SUCCESS:** Confirms that data has been successfully written to TimescaleDB.
  - **WRITE_FAILURE:** Indicates a failure in writing data to TimescaleDB.
  - **UPDATE_CACHE_SUCCESS:** Confirms successful updating of Redis cache.
  - **UPDATE_CACHE_FAILURE:** Signals a failure in updating Redis cache.
  - **RESET:** Resets the consumer from an error state back to idle.

- **Transitions:**
  - **Idle → Processing Message:** Triggered by the `MESSAGE_RECEIVED` event.
  - **Processing Message → Writing to DB:** Upon successful processing (`PROCESS_SUCCESS`).
  - **Processing Message → Error:** If processing fails (`PROCESS_FAILURE`).
  - **Writing to DB → Updating Redis:** Upon successful database write (`WRITE_SUCCESS`).
  - **Writing to DB → Error:** If database write fails (`WRITE_FAILURE`).
  - **Updating Redis → Idle:** Upon successful cache update (`UPDATE_CACHE_SUCCESS`).
  - **Updating Redis → Error:** If cache update fails (`UPDATE_CACHE_FAILURE`).
  - **Error → Idle:** Resetting after an error (`RESET`).

The **Consumer for Fixed Topics Actor** ensures data integrity and availability by accurately processing incoming tick data and maintaining up-to-date records in both the database and cache systems.

```
\documentclass{standalone}
\usepackage[svgnames]{xcolor} % Enables a wide range of color names
\usepackage{tikz}
\usetikzlibrary{positioning, shapes.geometric, arrows, automata}

\begin{document}

\begin{tikzpicture}[>=stealth, node distance=4cm, auto, scale=0.8, transform shape]

  % Define styles
  \tikzstyle{state} = [
    rectangle,
    rounded corners,
    minimum width=3cm,
    minimum height=1cm,
    draw,
    fill=LightCoral,
    font=\tiny
  ]
  \tikzstyle{trans} = [
    ->,
    >=stealth,
    thick,
    font=\tiny
  ]

  % Define states
  \node[state] (Idle) {Idle};
  \node[state, right=4cm of Idle] (ProcessingMessage) {Processing Message};
  \node[state, right=4cm of ProcessingMessage] (WritingToDB) {Writing to DB};
  \node[state, below right=2cm and 1cm of WritingToDB] (UpdatingCache) {Updating Redis};
  \node[state, below of=ProcessingMessage, yshift=-3cm, color=Red, fill=Red!20] (Error) {Error};

  % Define transitions
  \path[->, trans]
    (Idle) edge node {MESSAGE\_RECEIVED} (ProcessingMessage)
    (ProcessingMessage) edge node {PROCESS\_SUCCESS} (WritingToDB)
```

```
    (ProcessingMessage) edge node {PROCESS\_FAILURE} (Error)

    (WritingToDB) edge node {WRITE\_SUCCESS} (UpdatingCache)
    (WritingToDB) edge node {WRITE\_FAILURE} (Error)

    (UpdatingCache) edge node {UPDATE\_CACHE\_SUCCESS} (Idle)
    (UpdatingCache) edge node {UPDATE\_CACHE\_FAILURE} (Error)

    (Error) edge node {RESET} (Idle);

\end{tikzpicture}

\end{document}
```

Consumer for Dynamic Topics Actor Diagram

Tailored for handling application-specific data streams, this actor subscribes to dynamically created topics. It provides real-time OHLCV data to applications while leveraging caching for enhanced performance.

- **States:**
  - **Idle:** The consumer awaits commands to subscribe to new dynamic topics.
  - **Subscribing:** Initiates the subscription process to a newly created dynamic topic.
  - **Consuming:** Actively consumes messages from the subscribed dynamic topic.
  - **Processing Message:** Processes each incoming message to aggregate or format data as needed.
  - **Writing to DB:** Writes the processed data to TimescaleDB.
  - **Updating Redis:** Updates Redis with the latest data to ensure quick access for applications.
  - **Error:** Manages any issues that arise during subscription, message processing, database writing, or cache updating.
- **Events:**
  - **SUBSCRIBE_TO_TOPIC:** Commands the consumer to subscribe to a new dynamic topic.
  - **SUBSCRIBE_SUCCESS:** Confirms successful subscription to the dynamic topic.
  - **SUBSCRIBE_FAILURE:** Indicates a failure in subscribing to the dynamic topic.
  - **MESSAGE_RECEIVED:** Signals that a new message has been received from the dynamic topic.
  - **PROCESS_SUCCESS:** Confirms successful processing of the received message.
  - **PROCESS_FAILURE:** Signals a failure in processing the received message.
  - **WRITE_SUCCESS:** Confirms that data has been successfully written to TimescaleDB.
  - **WRITE_FAILURE:** Indicates a failure in writing data to TimescaleDB.
  - **UPDATE_CACHE_SUCCESS:** Confirms successful updating of Redis cache.
  - **UPDATE_CACHE_FAILURE:** Signals a failure in updating Redis cache.
  - **RESET:** Resets the consumer from an error state back to idle.
- **Transitions:**
  - **Idle Subscribing:** Triggered by the `SUBSCRIBE_TO_TOPIC` event.
  - **Subscribing Consuming:** Upon successful subscription (`SUBSCRIBE_SUCCESS`).
  - **Subscribing Error:** If subscription fails (`SUBSCRIBE_FAILURE`).
  - **Consuming Processing Message:** Triggered by the `MESSAGE_RECEIVED` event.
  - **Processing Message Writing to DB:** Upon successful processing (`PROCESS_SUCCESS`).
  - **Processing Message Error:** If processing fails (`PROCESS_FAILURE`).
  - **Writing to DB Updating Redis:** Upon successful database write (`WRITE_SUCCESS`).
  - **Writing to DB Error:** If database write fails (`WRITE_FAILURE`).
  - **Updating Redis Consuming:** Upon successful cache update (`UPDATE_CACHE_SUCCESS`).
  - **Updating Redis Error:** If cache update fails (`UPDATE_CACHE_FAILURE`).
  - **Error Idle:** Resetting after an error (`RESET`).

By subscribing to dynamic topics based on application requirements, the **Consumer for Dynamic Topics Actor** ensures that applications receive timely and relevant data streams without being burdened by the underlying data

infrastructure complexities.

```
\documentclass{standalone}
\usepackage[svgnames]{xcolor} % Enables a wide range of color names
\usepackage{tikz}
\usetikzlibrary{positioning, shapes.geometric, arrows, automata}

\begin{document}

\begin{tikzpicture}[>=stealth, node distance=4cm, auto, scale=0.8, transform shape]

  % Define styles
  \tikzstyle{state} = [
    rectangle,
    rounded corners,
    minimum width=3cm,
    minimum height=1cm,
    draw,
    fill=LightSeaGreen,
    font=\tiny
  ]
  \tikzstyle{trans} = [
    ->,
    >=stealth,
    thick,
    font=\tiny
  ]

  % Define states
  \node[state] (Idle) {Idle};
  \node[state, right=4cm of Idle] (Subscribing) {Subscribing};
  \node[state, right=4cm of Subscribing] (Consuming) {Consuming};
  \node[state, below right=2cm and 1cm of Consuming] (ProcessingMessage) {Processing Message};
  \node[state, below of=ProcessingMessage, yshift=-1cm] (WritingToDB) {Writing to DB};
  \node[state, below of=WritingToDB, yshift=-1cm] (UpdatingCache) {Updating Redis};
  \node[state, below of=Consuming, yshift=-3cm, color=Red, fill=Red!20] (Error) {Error};

  % Define transitions
  \path[->, trans]
    (Idle) edge node {SUBSCRIBE\_TO\_TOPIC} (Subscribing)
    (Subscribing) edge node {SUBSCRIBE\_SUCCESS} (Consuming)
    (Subscribing) edge node {SUBSCRIBE\_FAILURE} (Error)

    (Consuming) edge node {MESSAGE\_RECEIVED} (ProcessingMessage)

    (ProcessingMessage) edge node {PROCESS\_SUCCESS} (WritingToDB)
    (ProcessingMessage) edge node {PROCESS\_FAILURE} (Error)

    (WritingToDB) edge node {WRITE\_SUCCESS} (UpdatingCache)
    (WritingToDB) edge node {WRITE\_FAILURE} (Error)

    (UpdatingCache) edge node {UPDATE\_CACHE\_SUCCESS} (Consuming)
    (UpdatingCache) edge node {UPDATE\_CACHE\_FAILURE} (Error)

    (Error) edge node {RESET} (Idle);

\end{tikzpicture}
```

```
\end{document}
```

Datastore (API Service) Actor

The **Datastore** acts as the intermediary between external applications and the internal data platform. It abstracts the complexities of data retrieval and ensures that applications can seamlessly access both historical and real-time OHLCV data.

- **States:**
  - **Idle:** The datastore is awaiting API requests from applications.
  - **Processing Request:** Upon receiving a request, the datastore begins processing it.
  - **Querying DB:** Fetches the requested data from TimescaleDB.
  - **Data Found:** Confirms that the required data is available in the database.
  - **Data Not Found:** Indicates that the required data is missing from the database.
  - **Requesting Fetch:** Initiates a request for data fetching via the Coordinator and Producer actors.
  - **Waiting for Data:** Awaits the completion of data fetching and publishing.
  - **Returning Data:** Sends the combined historical and real-time data back to the requesting application.
  - **Error:** Manages any issues that arise during request processing, data querying, or data fetching.
- **Events:**
  - **REQUEST_DATA:** Triggered by an API call from an application requesting specific OHLCV data.
  - **START_DB_QUERY:** Initiates the process of querying TimescaleDB for historical data.
  - **DB_QUERY_SUCCESS:** Confirms that the required data has been successfully retrieved from the database.
  - **DB_QUERY_EMPTY:** Indicates that the requested data is not available in the database.
  - **DB_QUERY_FAILURE:** Signals a failure in querying the database.
  - **FETCH_INSTRUCTIONS_SENT:** Confirms that a request to fetch data has been sent to the Coordinator.
  - **DATA_FETCHED:** Confirms that new data has been successfully fetched and published.
  - **DATA_FETCH_FAILED:** Indicates a failure in fetching new data.
  - **RETURN_DATA_SUCCESS:** Confirms that data has been successfully returned to the application.
  - **RETURN_DATA_FAILURE:** Signals a failure in returning data to the application.
  - **RESET:** Resets the datastore from an error state back to idle.
- **Transitions:**
  - **Idle → Processing Request:** Triggered by the `REQUEST_DATA` event.
  - **Processing Request → Querying DB:** Upon initiating a database query (`START_DB_QUERY`).
  - **Querying DB → Data Found:** If data retrieval succeeds (`DB_QUERY_SUCCESS`).
  - **Querying DB → Data Not Found:** If no data is found (`DB_QUERY_EMPTY`).
  - **Querying DB → Error:** If database query fails (`DB_QUERY_FAILURE`).
  - **Data Found → Returning Data:** Direct transition to return data.
  - **Data Not Found → Requesting Data Fetch:** Triggered when data is missing.
  - **Requesting Data Fetch → Waiting for Data:** Upon sending fetch instructions (`FETCH_INSTRUCTIONS_SENT`).
  - **Waiting for Data → Returning Data:** When data is successfully fetched (`DATA_FETCHED`).
  - **Waiting for Data → Error:** If data fetch fails (`DATA_FETCH_FAILED`).
  - **Returning Data → Idle:** Upon successfully or unsuccessfully returning data (`RETURN_DATA_SUCCESS / RETURN_DATA_FAILURE`).
  - **Error → Idle:** Resetting after an error (`RESET`).

The **Datastore (API Service)** ensures that applications can request comprehensive datasets without needing to interact directly with the data ingestion or storage mechanisms. It intelligently merges historical data with real-time feeds, maintaining data integrity and continuity.

```
\documentclass{standalone}
\usepackage[svgnames]{xcolor} % Enables a wide range of color names
\usepackage{tikz}
\usetikzlibrary{positioning, shapes.geometric, arrows, automata}
```

```
\begin{document}

\begin{tikzpicture}[>=stealth, node distance=4.5cm, auto, scale=0.8, transform shape]

  % Define styles
  \tikzstyle{state} = [
    rectangle,
    rounded corners,
    minimum width=3.5cm,
    minimum height=1cm,
    draw,
    fill=LightYellow,
    font=\tiny
  ]
  \tikzstyle{trans} = [
    ->,
    >=stealth,
    thick,
    font=\tiny
  ]

  % Define states
  \node[state] (Idle) {Idle};
  \node[state, right=4.5cm of Idle] (ProcessingRequest) {Processing Request};
  \node[state, right=4.5cm of ProcessingRequest] (QueryingDB) {Querying DB};
  \node[state, below right=2cm and 0.5cm of QueryingDB] (DataFound) {Data Found};
  \node[state, below left=2cm and 0.5cm of QueryingDB] (DataNotFound) {Data Not Found};
  \node[state, below of=DataNotFound, yshift=-1cm] (RequestingFetch) {Requesting Data Fetch};
  \node[state, below of=RequestingFetch, yshift=-1cm, xshift=1cm] (WaitingForData) {Waiting for Data};
  \node[state, right=4.5cm of WaitingForData] (ReturningData) {Returning Data};
  \node[state, below of=ReturningData, yshift=-1cm, color=Red, fill=Red!20] (Error) {Error};

  % Define transitions
  \path[->, trans]
    (Idle) edge node {REQUEST\_DATA} (ProcessingRequest)

    (ProcessingRequest) edge node {START\_DB\_QUERY} (QueryingDB)

    (QueryingDB) edge node {DB\_QUERY\_SUCCESS} (DataFound)
    (QueryingDB) edge node {DB\_QUERY\_EMPTY} (DataNotFound)
    (QueryingDB) edge node {DB\_QUERY\_FAILURE} (Error)

    (DataFound) edge node [near start] {*} (ReturningData)

    (DataNotFound) edge node {*} (RequestingFetch)

    (RequestingFetch) edge node {FETCH\_INSTRUCTIONS\_SENT} (WaitingForData)

    (WaitingForData) edge node {DATA\_FETCHED} (ReturningData)
    (WaitingForData) edge node {DATA\_FETCH\_FAILED} (Error)

    (ReturningData) edge node {RETURN\_DATA\_SUCCESS / RETURN\_DATA\_FAILURE} (Idle)

    (Error) edge node {RESET} (Idle);

\end{tikzpicture}
```

```
\end{document}
```

Overall System Interactions

External Systems

- **TimescaleDB:** A time-series optimized database that stores aggregated OHLCV data. It leverages advanced features like continuous aggregates to automate data aggregation, ensuring scalability and efficiency.
- **Redis:** An in-memory data store used for caching recent OHLCV data. By caching frequently accessed data, Redis significantly reduces retrieval times, enhancing the responsiveness of the datastore to application requests.
- **Kafka/Redpanda:** The data streaming platform facilitating message passing between producers and consumers. It ensures reliable and scalable data dissemination across different components of the system.
- **CryptoCompare API:** The external data source providing both historical and real-time tick data. It supports RESTful endpoints for historical data and WebSocket connections for real-time streams, enabling the producer to fetch diverse data types as required.

Handling Real-Time Feeds and Data Aggregation

Real-Time Data Ingestion and Aggregation:

The system continuously ingests real-time tick data from CryptoCompare through the **Producer Actor**, publishing it to a fixed Redpanda topic, such as `tick-data`. The **Consumer for Fixed Topics Actor** subscribes to this topic, aggregating incoming tick data into OHLCV formats on 1-minute intervals. This aggregation can be efficiently handled by **TimescaleDB's continuous aggregates**, offloading the computational burden from consumers and ensuring scalability.

Implementing Aggregation Logic:

Aggregation can be accomplished through: - **Database-Level Aggregation:** Utilizing **TimescaleDB's continuous aggregates** to automatically compute OHLCV data from raw tick data, ensuring reliability and maintainability.

This setup allows the **Consumer for Fixed Topics Actor** to focus on data ingestion and ensures that the aggregation process benefits from database optimizations and scalability.

Caching with Redis:

To enhance data retrieval performance: - **Consumer Fixed Topics Actor** updates Redis with the latest OHLCV data after each aggregation cycle. - **Datastore** queries Redis first to fetch recent data, reducing latency and database load.

This strategy ensures that frequently accessed data is readily available, providing quick responses to application requests.

Integrating Historical Data with Real-Time Feeds

When an application requests historical OHLCV data from a specific start date, the **Datastore** performs the following steps to ensure seamless integration with real-time data streams:

1. **Determine the Last Available Timestamp:**
   - The datastore queries **TimescaleDB** to identify the latest timestamp for the requested symbol and timeframe.
2. **Fetch Historical Data:**
   - Retrieves all OHLCV data from **TimescaleDB** starting from the user-specified date up to the last available timestamp.
3. **Align Real-Time Data:**
   - Ensures that the real-time data feed begins immediately after the last historical data point, maintaining continuity and eliminating overlaps or gaps.
4. **Merge Data Streams:**

- Combines the fetched historical data with the incoming real-time data, presenting a cohesive dataset to the application.

5. **Handle Data Gaps:**
   - Implement mechanisms to verify that there are no discontinuities between historical and real-time data, ensuring data integrity.

By intelligently managing the transition between historical and real-time data, the **Datastore** provides applications with comprehensive and continuous datasets without exposing the complexities of underlying data flows.

Error Handling and Resilience

Each actor within the system is designed with robust error handling mechanisms to ensure resilience and maintain uninterrupted data flow:

- **Error States:** Every component includes an error state that captures and manages failures, whether they occur during data fetching, publishing, processing, or caching.
- **Transitions to Error States:** Upon encountering an issue, actors transition to their respective error states, allowing for controlled handling and recovery.
- **Reset Mechanisms:** From error states, actors can transition back to idle or operational states upon receiving reset commands, facilitating recovery and continuation of normal operations.
- **Logging and Monitoring:** Comprehensive logging within each state and transition aids in monitoring system health and diagnosing issues promptly.

System Scalability and Flexibility

The architecture is inherently scalable and flexible, accommodating growing data volumes and evolving application requirements:

- **Dynamic Topic Management:** The **Coordinator Actor** can create and manage dynamic topics on-demand, enabling the system to adapt to various data consumption needs without manual intervention.
- **Modular Actors:** Each actor operates independently, allowing for horizontal scaling. For instance, multiple instances of **Consumer Actors** can handle increased data loads by subscribing to the same or different topics.
- **Efficient Data Storage:** Leveraging **TimescaleDB** for time-series data and **Redis** for caching ensures optimized storage and retrieval performance, even as data scales.
- **Seamless Integration with Applications:** The **Datastore (API Service)** abstracts the complexities of data ingestion and storage, providing a straightforward interface for applications to access the required data without needing to interact directly with streaming platforms or databases.

Overall System Interaction Diagram

```
\documentclass{standalone}
\usepackage[svgnames]{xcolor} % Enables a wide range of color names
\usepackage{tikz}
\usetikzlibrary{positioning, shapes.geometric, arrows, automata, fit}

\begin{document}

\begin{tikzpicture}[>=stealth, node distance=4.5cm, auto, scale=0.7, transform shape]

  % Define styles
  \tikzstyle{actor} = [
    rectangle,
    rounded corners,
    minimum width=3cm,
    minimum height=1cm,
    draw,
```

```
  fill=LightBlue,
  font=\tiny
]
\tikzstyle{external} = [
  ellipse,
  minimum width=2.5cm,
  minimum height=1cm,
  draw,
  fill=LightGrey,
  font=\tiny
]
\tikzstyle{dataStore} = [
  rectangle,
  rounded corners,
  minimum width=4cm,
  minimum height=1.5cm,
  draw,
  fill=LightYellow,
  font=\tiny
]
\tikzstyle{db} = [
  ellipse,
  minimum width=2.5cm,
  minimum height=1cm,
  draw,
  fill=LightGrey,
  font=\tiny
]
\tikzstyle{cache} = [
  ellipse,
  minimum width=2.5cm,
  minimum height=1cm,
  draw,
  fill=LightGrey,
  font=\tiny
]
\tikzstyle{trans} = [
  ->,
  >=stealth,
  thick,
  font=\tiny
]

% External Systems
\node[external] (CryptoCompare) {CryptoCompare API};
\node[external, below of=CryptoCompare, yshift=-1cm] (Applications) {Applications};

% Actors
\node[actor, below of=CryptoCompare, yshift=-3cm] (Producer) {Producer Actor};
\node[actor, right=6cm of Producer] (Coordinator) {Coordinator Actor};
\node[actor, below of=Producer, yshift=-3cm] (ConsumerFixed) {Consumer Fixed Topics Actor};
\node[actor, below of=Coordinator, yshift=-3cm] (ConsumerDynamic) {Consumer Dynamic Topics Actor};

% Datastore and Infrastructure
\node[dataStore, below of=ConsumerFixed, yshift=-3cm] (Datastore) {Datastore (API Service)};
\node[db, right=3cm of Datastore] (TimescaleDB) {TimescaleDB};
```

```
  \node[cache, below of=TimescaleDB, yshift=-1cm] (Redis) {Redis};

  % Kafka Cluster
  \node[rectangle, draw, dashed, fit=(Producer) (Coordinator) (ConsumerFixed) (ConsumerDynamic),
   ↪  label=above:{Kafka/Redpanda Cluster}, inner sep=0.5cm] (KafkaCluster) {};

  % Connections

  % Producer to External API
  \path[->, trans]
    (Producer) edge node {Fetch Data} (CryptoCompare)
    (CryptoCompare) edge node {Return Data} (Producer);

  % Producer to Kafka Cluster
  \path[->, trans]
    (Producer) edge node {Publish to Fixed Topics} (ConsumerFixed)
    (Producer) edge node {Publish to Dynamic Topics} (ConsumerDynamic);

  % Coordinator to Producer
  \path[->, trans]
    (Coordinator) edge node {Instruct to Create Topic} (Producer);

  % Coordinator to Consumer Dynamic
  \path[->, trans]
    (Coordinator) edge node {Manage Dynamic Subscribers} (ConsumerDynamic);

  % Consumer Fixed to TimescaleDB and Redis
  \path[->, trans]
    (ConsumerFixed) edge node {Write OHLCV to DB} (TimescaleDB)
    (ConsumerFixed) edge node {Update Redis Cache} (Redis);

  % Consumer Dynamic to TimescaleDB and Redis
  \path[->, trans]
    (ConsumerDynamic) edge node {Write OHLCV to DB} (TimescaleDB)
    (ConsumerDynamic) edge node {Update Redis Cache} (Redis);

  % Applications to Datastore
  \path[->, trans]
    (Applications) edge node {Request Data} (Datastore);

  % Define edges with corrected labels
  \path[->, trans]
  (Datastore) edge node {Query DB} (TimescaleDB)
  (Datastore) edge node {Check Cache} (Redis)
  (Redis) edge [bend left=45, dashed] node [right] {Cache Hit} (Datastore)
  (Redis) edge [bend left=45, dashed] node [right] {Cache Miss} (Datastore);

\end{tikzpicture}

\end{document}
```

Diagram Explanation

- **External Systems:**
  - **CryptoCompare API:** Source of both historical and real-time tick data.
  - **Applications:** External applications requesting data from the Datastore.

- **Actors:**
  - **Producer Actor:** Fetches data from CryptoCompare and publishes to Redpanda topics.
  - **Coordinator Actor:** Manages dynamic topic creation and orchestrates interactions between Producers and Consumers.
  - **Consumer Fixed Topics Actor:** Consumes fixed topic data, aggregates into OHLCV, writes to TimescaleDB, and updates Redis.
  - **Consumer Dynamic Topics Actor:** Consumes dynamically created topics, aggregates into OHLCV, writes to TimescaleDB, and updates Redis.
- **Datastore (API Service):**
  - Acts as an intermediary API layer for Applications to request OHLCV data.
  - Interfaces with **TimescaleDB** and **Redis** for data retrieval.
  - Decides whether to serve data from Redis, TimescaleDB, or fetch new data via the Coordinator and Producer Actors.
- **Infrastructure Components:**
  - **TimescaleDB:** Stores aggregated OHLCV data with time-series optimizations.
  - **Redis:** Provides in-memory caching for recent OHLCV data to expedite retrieval.
- **Kafka/Redpanda Cluster:**
  - Facilitates message passing between Producers and Consumers.
  - Represented as a dashed rectangle encapsulating the Producer, Coordinator, and Consumer actors.
- **Connections:**
  - **Data Flow:**
    - **Producer CryptoCompare:** Fetches data.
    - **CryptoCompare Producer:** Returns fetched data.
    - **Producer ConsumerFixed / ConsumerDynamic:** Publishes data to Redpanda topics.
    - **Consumers TimescaleDB:** Write aggregated OHLCV data.
    - **Consumers Redis:** Update caches.
  - **Application Requests:**
    - **Applications Datastore:** Sends data requests.
    - **Datastore Redis:** Checks cache.
    - **Redis Datastore:** Returns cached data or indicates a cache miss.
    - **Datastore TimescaleDB:** Queries historical data.
    - **Datastore Coordinator:** Initiates data fetch if necessary.
  - **Dynamic Topic Management:**
    - **Coordinator Producer:** Instructs to create dynamic topics.
    - **Coordinator ConsumerDynamic:** Manages dynamic subscribers.
- **Annotations:**
  - **Dashed Lines & Arrows:** Represent data flow and interactions between components.
  - **Color Coding:** Differentiates between types of components and their states.

Conclusion

The data platform architecture is designed to handle both historical and real-time data efficiently. By defining the roles and responsibilities of each actor, implementing robust error handling, and ensuring scalability through dynamic topic management and optimized storage solutions, we hope that the system is well-equipped to meet diverse data processing and application consumption needs. The **Datastore (API Service)** plays a pivotal role in bridging applications with the data platform, ensuring seamless data access and integrity.

As we proceed with implementing each component, maintaining clear boundaries and communication channels between actors will be crucial. Regular monitoring and iterative testing will further ensure that the system remains resilient and performs optimally under varying workloads.