A Distributed Online Learning Approach for Pattern Prediction over Movement Event Streams with Apache Flink

Ehab Qadah
Fraunhofer IAIS
Sankt Augustin, Germany
Ehab.Qadah@iais.fraunhofer.de

Michael Mock
Fraunhofer IAIS
Sankt Augustin, Germany
Michael.Mock@iais.fraunhofer.de

Elias Alevizos
NCSR "Demokritos"
Athens, Greece
alevizos.elias@iit.demokritos.gr

Georg Fuchs
Fraunhofer IAIS
Sankt Augustin, Germany
Georg.Fuchs@iais.fraunhofer.de

ABSTRACT

In this paper, we present a distributed online prediction system for user-defined patterns over multiple massive streams of movement events, built using the general purpose stream processing framework Apache Flink. The proposed approach is based on combining probabilistic event pattern prediction models on multiple predictor nodes with a distributed online learning protocol in order to continuously learn the parameters of a global prediction model and share them among the predictors in a communication-efficient way. Our approach enables the collaborative learning between the predictors (i.e., "learn from each other"), thus the learning rate is accelerated with less data. The underlying model provides online predictions about when a pattern (i.e., a regular expression over the event types) is expected to be completed within each event stream. We describe the distributed architecture of the proposed system, its implementation in Flink, and present experimental results over real-world event streams related to trajectories of moving vessels.

KEYWORDS

Big Movement Event Streams, Stream Processing, Event Pattern Prediction, Distributed Pattern Markov Chain.

1 INTRODUCTION

In recent years, technological advances have led to a growing availability of massive amounts of continuous streaming data (i.e., data streams observing events) in many application domains such as social networks [**?**], Internet of Things (IoT) [**?**], and maritime surveillance [**?**]. The ability to detect and predict the full matches of a pattern of interest (e.g., certain sequence of events), defined by a domain expert, is important for several operational decision making tasks.

An event stream is an unbounded collection of timely ordered data observations in the form of an attribute tuple that is composed of a value from finite event types along with other categorical and numerical attributes. In this work we deal with movement event streams. For instance, in the context of maritime surveillance, the event stream of a moving vessel consists of spatiotemporal and kinematic information along with the vessel's identification and its trajectory related events, based on the automatic identification system (AIS) [**?**] messages that are continuously sent by the vessel. Therefore, leveraging event patterns prediction over real-time streams of moving vessels is useful to alert maritime operation managers about suspicious activities (e.g., fast sailing vessels near ports, or illegal fishing) before they happen. However, processing real-time streaming data with low latency is challenging, since data streams are large and distributed in nature and continuously arrive at a high rate.

In this paper, we present the design and implementation of an online, distributed and scalable pattern prediction system over multiple, massive streams of events. More precisely, we consider event streams related to trajectories of moving objects (i.e., vessels). The proposed approach is based on a novel method that combines a distributed online prediction protocol [**? ?**] with an event forecasting method, based on Markov chains [**?**]. It is implemented on top of the Big Data framework for stream processing Apache Flink [**?**]. We evaluate our proposed system over real-word data streams of moving vessels, which are provided in the context of the datAcron project[1].

The rest of the paper is organized as follows. We discuss the related work and used frameworks in Section 2. In Section 3, we describe the problem of pattern prediction, our proposed approach, and the architecture of our system. The implementation details on top of Flink are presented in Section 4 and the experimental results in Section 5. We conclude in Section 6.

[1] http://www.datacron-project.eu/

2 RELATED WORK AND BACKGROUND

2.1 Related work

Pattern prediction over event streams. The task of forecasting over time-evolving streams of data can be formulated in various different ways and with varying assumptions. One of the most usual ways to formalize this task is to assume that the stream is a time-series of numerical values and the goal is to forecast at each time point n the values at some future points $n+1$, $n+2$, etc., (or even the output of some function of future values). This is the task of time-series forecasting [**?**]. Another way to formalize this task is to view streams as sequences of events, i.e., tuples with multiple, possibly categorical, attributes, like *event type*, *timestamp*, etc., and the goal is to predict future events or patterns of events. In the work that we present here, we focus on this latter definition of forecasting (event pattern forecasting).

A substantial body of work on event forecasting comes from the field of temporal pattern mining where events are defined as 2-tuples of the form (*EventType*, *Timestamp*). The ultimate goal is to extract patterns of events in the form either of association rules [**?**] or frequent episode rules [**?**]. These methods have been extended in order to be able to learn not only rules for detecting event patterns but also rules for predicting events. For example, in [**?**], a variant of association rule mining is where the goal is to extract sets of event types that frequently lead to a rare, target event within a temporal window.

In [**?**], a probabilistic model is presented in order to calculate the probability of the immediately next event in the stream. This is achieved by using standard frequent episode discovery algorithms and combining them with Hidden Markov Models and mixture models. The framework of episode rules is employed in [**?**] as well. The output of the proposed algorithms is a set of predictive rules whose antecedent is minimal (in number of events) and temporally distant from the consequent. In [**?**] a set of algorithms is proposed that target batch, online mining of sequential patterns, without maintaining exact frequency counts. As the stream is consumed, the learned patterns can be used to test whether a prefix matches the last events seen in the stream, indicating a possibility of occurrence for events that belong to the suffix of the rule.

Event forecasting has also attracted some attention from the filed of Complex Event Processing (see [**?**] for a review of Complex Event Processing). One such early approach is presented in [**?**]. Complex event patterns are converted to automata and subsequently Markov chains are used in order to estimate when a pattern is expected to be fully matched. A similar approach is presented in [**?**], where again automata and Markov chains are employed in order to provide (future) time intervals during which a match is expected with a probability above a confidence threshold.

Distributed Online Learning. In recent years, the problem of distributed online learning has received significant attention and has been studied in [**? ? ? ? ?**]. A distributed online mini-batch prediction approach over multiple data streams has been proposed in [**?**]. This approach is based on a static synchronization method. The learners periodically communicate their local models with a central coordinator unit after consuming a fixed number of input samples/events (i.e., batch size b), in order to create a global model and share it between all learners. This work has been extended in [**?**] by introducing a dynamic synchronization scheme that reduces the required communication overhead. It can do so by making the local learners communicate their models only if they diverge form a reference model. In this work, we aim to employee this protocol with event patterns prediction models over multiple event streams.

2.2 Technological Background

In the last years, many systems for large-scale and distributed stream processing have been proposed, including Spark Streaming [**?**], Apache Storm [**?**] and Apache Flink [**?**]. These frameworks can ingest and process real-time data streams, published from different distributed message queuing platforms, such as Apache Kafka [**?**] or Amazon Kinesis [**?**]. In this work, we implemented the proposed system over Apache Flink. Flink provides the distributed stream processing components of the distributed event pattern predictors. It works alongside Apache kafka, which is employed for streaming the input event streams and as a messaging platform to enable the distributed online learning functionalities.

In the datAcron project, the Flink streaming processing engine has been chosen as a primary platform for supporting the streaming operations, based on an internal comparative evaluation of several streaming platforms. Thus in this work we used it to implement our system. Although the distributed online learning framework has already been implemented in the FERARI project [**?**] based on Apache Storm.

Apache Flink. Apache Flink is an open source project that provides a large-scale, distributed and stateful stream processing platform [**?**]. Flink is one of the most recent and pioneering big data processing frameworks. It provides processing models for both streaming and batch data, where the batch processing model is treated as a special case of the streaming one (i.e., finite stream). Flink's software stack includes the *DataStream* and *DataSet* APIs for processing infinite and finite data, respectively. These two core APIs are built on top of Flink's core dataflow engine and provide operations on data streams or sets such as mapping, filtering, grouping, etc.

The main data abstractions of Flink are *DataStream* and *DataSet* that represent read-only collections of data elements. The list of elements is bounded (i.e., finite) in *DataSet*, while it is unbounded (i.e., infinite) in the case of

DataStream. Flink's core is a distributed streaming dataflow engine, where each Flink program is represented by a dataflow graph (i.e., directed acyclic graph - DAG) that gets executed by Flink's engine [**?**]. The data flow graphs are composed of stateful operators and intermediate data stream partitions. The execution of each operator is handled by multiple parallel instances whose number is determined by the *parallelism* level. Each parallel operator instance is executed in an independent task slot on a machine within a cluster of computers [**?**].

Apache Kafka. Apache Kafka is a scalable, fault-tolerant, and distributed streaming framework/messaging system [**?**]. It allows to publish and subscribe to arbitrary data streams, which are managed in different categories (i.e., *topics*) and partitioned in the Kafka cluster. The Kafka Producer API provides the ability to publish a stream of messages to a topic. These messages can then be consumed by applications, using the Consumer API that allows them to read the published data stream in the Kafka cluster. In addition, the streams of messages are distributed and load balanced between the multiple receivers within the same consumer group for the sake of scalability.

3 SYSTEM OVERVIEW

3.1 Pattern prediction on a single stream

For our work presented in this paper, we use the approach described in [**?**]. For the sake of self-containment, we briefly describe this approach in what follows, first assuming that only a single stream is consumed and then adjusting for the case of multiple streams. We follow the terminology of [**? ? ?**] to formalize the problem we tackle.

3.1.1 Problem formulation. We first give the definition for an input event and for a stream of input events as follows:

Definition 3.1. Each event is defined as a tuple of attributes $e_i = (id, type, \tau, a_1, a_2 ....., a_n)$, where *type* is the event type attribute that takes a value from a set of finite event types/symbols Σ, τ represents the time when the event tuple was created, the $a_1, a_2, ..., a_n$ are spatial or other contextual features (e.g., speed); these features are varying from one application domain to another. The attribute *id* is a unique identifier that connects the event tuple to an associated domain object.

Definition 3.2. A stream $s = \langle e_1, e_3, ..., e_t, ... \rangle$ is a time-ordered sequence of events.

A user-defined pattern $\mathcal{P}$ is given in the form of a regular expression (i.e., using operators for *sequence*, *disjunction*, and *iteration*) over Σ (i.e., event types) [**?**]. More formally, a pattern is given through the following grammar:

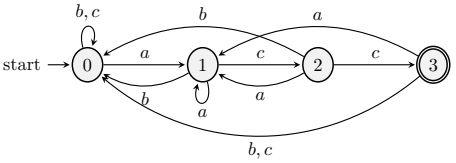
Definition 3.3. $\mathcal{P} := E \mid \mathcal{P}_1; \mathcal{P}_2 \mid \mathcal{P}_1 \vee \mathcal{P}_2 \mid \mathcal{P}_1^*$, where $E \in \Sigma$ is a constant event type, ; stands for sequence, $\vee$ for disjunction and $*$ for *Kleene* $-*$. The pattern $\mathcal{P} := E$ is matched by reading an event e_i iff $e_i.type = E$. The other cases are matched as in standard automata theory.

The problem at hand may then be stated as follows: given a stream s of low-level events and a pattern $\mathcal{P}$, the goal is to estimate at each new event arrival the number of future events that we will need to wait for until the pattern is satisfied (and therefore a match be detected).

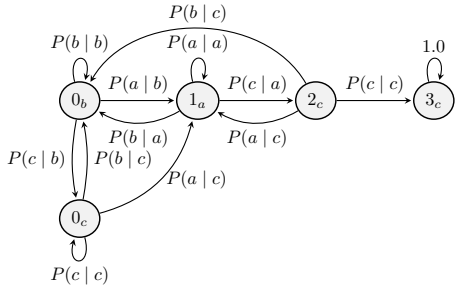
3.1.2 Proposed approach. As a first step, event patterns are converted to deterministic finite automata (DFA) through standard conversion algorithms. As an example, see Figure 1a for the DFA of the simple sequential pattern $\mathcal{P} = a; c; c$ and an alphabet $\Sigma = \{a, b, c\}$ (note that the DFA has no dead states since we need to handle streams and not strings). The next step is to derive a Markov chain that will be able to provide a probabilistic description of the DFA's run-time behavior. Towards this goal, we use Pattern Markov Chains, as was proposed in [**?**]. Under the assumption that the input events are independent and identically distributed (i.i.d.), it can be shown that there is a direct mapping of the states of the DFA to states of a Markov chain and the transitions of the DFA to transitions of the Markov chain. The transition probabilities of the Markov chain are the occurrence probabilities of the various event types. On the other hand, if the occurrence probabilities of the events are dependent on some of the previous events seen in the stream (i.e., the stream is generated by an m^{th} order Markov process), we might need to perform a more complex transformation (see [**?**] for details) in order to obtain a "proper" Markov chain. The transition probabilities are then conditional probabilities on the event types. In any case, we call such a derived Markov chain a Pattern Markov Chain (PMC) of order m and denote by $PMC_{\mathcal{P}}^m$, where $\mathcal{P}$ is the initial pattern and m the assumed order. As an example, see Figure 1b, which depicts the PMC of order 1 for the generated DFA of Figure 1a.

After constructing a PMC, we can use it in order to calculate the so-called *waiting-time* distributions. Given a specific state of the PMC, a *waiting-time* distribution gives us the probability of reaching a set of absorbing states in n transition from now (absorbing states are states with self-loops and probability equal to 1.0). By mapping the final states of the initial DFA to absorbing states of the PMC (see again Figure 1), we can therefore calculate the probability of reaching a final state, or, in other words, of detecting a full match of the original regular expression in n events from now.

In order to estimate the final forecasts, another step is required, since our aim is not to provide a single future point with the highest probability but an interval. Predictions are given in the form of intervals, like $I = (start, end)$. The meaning of such an interval is that the DFA is expected to reach a final state sometime in the future between *start* and *end* with probability at least some constant threshold θ_{fc} (provided by the user). These intervals are estimated by a



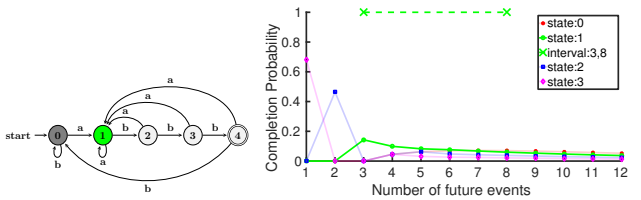
(a) $DFA_{\Sigma^* \cdot \mathcal{P}}$



(b) $PMC^1_{\mathcal{P}}$

Figure 1: DFA and PMC for $\mathcal{P} = a; c; c$, $\Sigma = \{a, b, c\}$, and order $m = 1$ [?].

single-pass algorithm that scans a waiting-time distribution and finds the smallest (in terms of length) interval that exceeds this threshold. An example is shown in Figure 2, where the DFA in Figure 2a is in state 1, the *waiting-time* distributions for all of its non-final states are shown in Figure 2b and the distribution, along with the prediction interval, for state 1 are shown in green.



(a) DFA, state 1. **(b) Waiting-time distribution, state 1.**

Figure 2: Example of how prediction intervals are produced. $\mathcal{P} = a; b; b; b$, $\Sigma = \{a, b\}$, $m = 1$, $\theta_{fc} = 0.5$ [?].

The above described method assumes that we know the (possibly conditional) occurrence probabilities of the various event types appearing in a stream (as would be the case with synthetically generated streams). However, this is not always the case in real-world situations. Therefore, it is crucial for a system implementing this method to have the capability to learn the values of the PMC's transition matrix. One way to do this is to use some part of the stream to obtain the maximum-likelihood estimators for the transition probabilities. If $\mathbf{\Pi}$ is the transition matrix of a Markov chain with a set of states Q, $\pi_{i,j}$ the transition probability from state i to state j, $n_{i,j}$ the number of transitions from state i to state j, then the maximum likelihood estimator for $\pi_{i,j}$ is given by:

$$\hat{\pi}_{i,j} = \frac{n_{i,j}}{\sum_{k \in Q} n_{i,k}} = \frac{n_{i,j}}{n_i}$$

Executing this learning step on a single node might require a vast amount of time until we arrive at a sufficiently good model. In this paper, we present a distributed method for learning the transition probability matrix.

3.2 Pattern prediction on multiple streams

3.2.1 Problem formulation. Let $O = \{o_1, ..., o_k\}$ be a set of K objects (i.e., moving objects) and $S = \{s_1, ..., s_k\}$ a set of real-time streams of events, where s_i is generated by the object o_i. Let $\mathcal{P}$ be a user-defined pattern which we want to apply to every stream s_i, i.e., each object will have its own DFA.

The setting that is considered in this work is then described in the following: we have multiple input event streams and a system consisting of K distributed predictor nodes $n_1, n_2..., n_k$, each of which consumes an input event stream $s_i \in S$. The goal is to provide timely predictions and be able to do this in scale. Each node n_i $i \in [K]$ handles a single event stream s_i associated with a moving object $o_i \in O$. In addition, it maintains a local prediction model f_i for the user-defined pattern $\mathcal{P}$. The f_i model provides the online prediction about the future full match of the pattern $\mathcal{P}$ in s_i for each new arriving event tuple. In short, we have multiple running instances of an online prediction algorithm on distributed nodes for multiple input event streams. More specifically, the input to our system consists of massive streams of events that describe trajectories of moving vessels in the context of maritime surveillance, where there is one predictor node for each vessel's event stream.

3.2.2 The proposed approach. We design and develop a scalable and distributed patterns prediction system over massive input event streams of moving objects. As the base prediction model, we use the PMC forecasting method [?]. Moreover, we propose to enable the information exchange between the distributed predictors/learners of the input event streams, by adapting the distributed online prediction protocol of [?] to synchronize the prediction models, i.e., the transitions probabilities matrix of the PMC predictors.

Algorithm 1 presents the distributed online prediction protocol by dynamic model synchronization on both the predictor nodes and the coordinator. We refer to the PMC's transition matrix $\mathbf{\Pi}_i$ on predictor node n_i by f_i. That is, when a predictor $n_i : \ i \in [k]$ observes an event e_j it revises its internal model state (i.e., f_i) and provides a prediction report. Then it checks the local conditions (batch size b and local model divergence from a reference model f_r) to decide if there is a need to synchronize its local model with the coordinator or not. f_r is maintained in the predictor node as a copy of the last computed aggregated model $\hat{f}$

from the previous full synchronization step, which is shared between all local predictors/learners. By monitoring the local condition $\|f_i - f_r\|^2 > \Delta$ on all local predictors, we have a guarantee that if none of the local conditions is violated, the divergence (i.e., variance of local models $\delta(f) = \frac{1}{k}\sum_{j=1}^{k}\|f_i - \hat{f}\|^2$) does not exceed the threshold Δ [**?**].

On the other hand, the coordinator receives the prediction models from the predictor nodes that requested for model synchronization (violation), then tries to keep incrementally querying other nodes for their local prediction models until reaching out all nodes, or the variance of the aggregated model computed from the already received models less or equal than the divergence threshold Δ. Finally, an aggregated model $\hat{f}$ is computed and sent back to the predictor nodes that sent their models after violation or have been queried by the coordinator.

Algorithm 1: Communication-efficient Distributed Online Learning Protocol

Predictor node n_i: at observing event e_j
  update the prediction model parameters f_i and provide a prediction service
  if $j \mod b = 0$ *and* $\|f_i - f_r\|^2 > \Delta$ **then**
    send f_i to the Coordinator (violation)

Coordinator:
  receive local models with violation $B = \{f_i\}_{i=1}^{m}$
  while $|B| \neq k$ *and* $\frac{1}{|B|}\sum_{f_i \in \Pi}\|f_i - \hat{f}\|^2 > \Delta$ **do**
    add other nodes have not reported violation for their models $B \leftarrow \{f_l : f_l \notin B \text{ and } l \in [k]\}$
    receive models from nodes add to B
  compute a new global model $\hat{f}$
  send $\hat{f}$ to all the predictor nodes in B and set $f_1 \ldots, f_m = \hat{f}$
  if $|B| = k$ **then**
    set a new reference model $f_r \leftarrow \hat{f}$

This protocol was introduced for linear models, and has been extended to handle kernelized online learning models [**?**]. We also employ this protocol for the pattern prediction model, which is internally based on the PMC $PMC_{\mathcal{P}}^{m}$. This allows the distributed $PMC_{\mathcal{P}}^{m}$ predictors for multiple event streams to synchronize their models (i.e., transition probability matrix of each predictor) within the system in a communication-efficient manner.

We propose a *synchronization operation* for the models parameters ($f_i = \mathbf{\Pi}_i : i \in [k]$) of the k distributed PMC predictors. The operation is based on distributing the maximum-likelihood estimation [**?**] for the transition probabilities of the underlying $PMC_{\mathcal{P}}^{m}$ models described by:

$$\hat{\pi}_{i,j} = \frac{\sum_{k \in K} n_{k,i,j}}{\sum_{k \in K}\sum_{l \in L} n_{k,i,l}}$$

Moreover, we measure the divergence of local models from the reference model $\|f_k - f_r\|^2$ by calculating the sum of square difference between the transition probabilities $\mathbf{\Pi}_i$ and $\mathbf{\Pi}_r$:

$$\|f_k - f_r\|^2 = \sum_{i,j}(\hat{\pi}_k i, j - \hat{\pi}_r i, j)^2$$

In general, our approach relies on enabling the collaborative learning between the prediction models of the input event streams. By doing so, we assume that the underlying event streams belong to the same distribution and share the same behavior (e.g., mobility patterns). We claim that assumption is reasonable in many application domains. For instance, in the context of maritime surveillance, vessels travel through standard routes, defined by the International Maritime Organization (IMO). Additionally, vessels have similar mobility patterns in specific areas such as moving with low speed and multiple turns near the ports [**? ?**]. That allows our system to dynamically construct a coherent global prediction model for all input event streams based on merging their local prediction models.

3.3 Distributed architecture

Our system consumes as input[2] an aggregated stream of events coming from a large number of moving objects, which is continuously collected and fed into the system. It allows users to register a pattern $\mathcal{P}$ to be monitored over each event stream of a moving object. The output stream consists of original input events and predictions of full matches of $\mathcal{P}$, displayed to the end users. Figure 3 presents the overview of our system architecture and its main components.

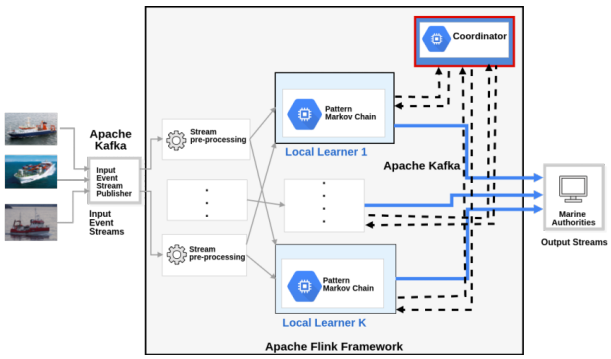


Figure 3: System Architecture.

The system is composed of three processing units: (i) pre-processing operators that receive the input event stream and perform filtering and ordering operations, before partitioning the input event stream to multiple event streams based on the associated moving object (ii) predictor nodes (learners), which are responsible for maintaining a prediction model for the input event streams. Each prediction node is configured to handle an event stream from the same moving

[2]In practice, the aggregated input events stream is composed of multiple event streams (partitions) for multiple moving objects, which are reconstructed by the system internally.

object, in order to provide online predictions for a predefined pattern $\mathcal{P}$ (iii) a coordinator node that communicates through Kafka stream channels with the predictors to realize the distributed online learning protocol. It builds a global prediction model, based on the received local models, and then shares it among the predictors.

Our distributed system consists of multiple pre-processing operators, prediction nodes and a central coordinator node. All units run concurrently and are arranged as a data processing pipeline, depicted in Figure 3. We leverage Apache Kafka as a messaging platform to ingest the input event streams and to publish the resulting streams. In addition, it is used as the communication channel between the predictor nodes and the coordinator. Apache Flink is employed to execute the system's distributed processing units over the input event streams: the pre-processing unit, the prediction unit and the coordinator unit. Our system architecture can be modeled as a logical network of processing nodes, organized in the form of a DAG, inspired by the Flink runtime dataflow programs [**?**].

4 IMPLEMENTATION DETAILS

This section provides a detailed overview of the implementation of our system, it has been implemented on top of Apache Flink and Apache Kafka frameworks. Each of the three sub-modules described in Section 3.3 have been implemented as Flink operations over the input events stream.

Pre-processing and Prediction Operators. Listing 1 shows how the main workflow of the system is implemented as Flink data flow program.

```
DataStream<Event> inputEventsStream =
     env.addSource(flinkKafkaConsumer);
// Create event tuples (id,event) and assign time
     stamp
DataStream<Tuple2<String, sEvent>>
     eventTuplesStream =
inputEventsStream.map(new EventTuplesMapper())
.assignTimestampsAndWatermarks(new
     EventTimeAssigner());
// Create the ordered keyed stream
KeyedStream<Tuple2<String, Event>, Tuple>
     keyedEventsStream =
eventsStream.keyBy(0).process(new EventSorter())
.keyBy(0);
//Initialize the predictor node
LocalPredictorNode predictorNode =new
     LocalPredictorNode<Event>(P);
// Process the keyedEventsStream by the predictor
DataStream<Event> processedEventsStream =
keyedEventsStream.map(predictorNode);
```

Listing 1: Flink pipeline for local predictors workflow

The system ingests the input events stream form a Kafka cluster that is mapped to a *DataStream* of events, which is then processed by a *EventTuplesMapper* in order to create tuples of *(id,eventObject)*, where the *id* is associated to the identifier of the moving object. To handle events coming in out of order in a certain margin, the stream of event tuples is processed by a *TimestampAssigner*, it assigns the timestamps for the input events based on the extracted creation time. Afterwards, an ordered stream of event tuples is generated using a process function *EventSorter*. The ordered stream is then transformed to a *keyedEventsStream* by partitioning it based on the ids values using a *keyBy* operation. A local *predictor* node in a distributed environment is represented by a *map* function over the *keyedEventsStream*, while each parallel instance of the map operator (predictor) always processes all events of the same moving object (i.e., equivalent id), and maintains a bounded prediction model (i.e., $PMC^m_{\mathcal{P}}$ predictor) using the Flink's Keyed State [3]. The output streams of the moving objects form the parallel instances of the predictor map functions are sent to a new Kafka stream (i.e., same topic name). They then can be processed by other components like visualization or users notifier.

Moreover, the implementation of the *predictor* map function includes the communication with *coordinator* using Kafka streams. At the beginning of the execution it sends a registration request to the coordinator. Also through the execution of the system over the input event stream, it sends its local prediction model as synchronization request, or as a response for a resolution request from the coordinator. These communication messages are published into different Kafka topics as depicted in Table 1.

Table 1: Messages to Kafka topics mapping.

Message	Kafka Topic
RegisterNode, **RequestSync** and **ResolutionAnswer**	LocalToCoordinatorTopicId
CoordinatorSync and **RequestResolution**	CoordinatorToLocalTopicId

Coordinator. Listing 2 presents the workflow of the coordinator node that manages the distributed online learning protocol operations, which is implemented as Flink program. The coordinator receives messages from the local predictors through a Kafka Stream of a topic named *"LocalToCoordinatorTopicId"*. It is implemented as a single *map* function over the messages stream, by setting the *parallelism* level of the Flink program to *"1"*. Increasing the parallelism will scale up the number of parallel coordinator instances, for example, in order to handle different groupings of the input event streams. The map operator of the coordinator handles three message types from the predictors: (i) **RegisterNode** that contains a registration

[3] https://ci.apache.org/projects/flink/flink-docs-release-1.3/dev/stream/state.html#kayed-state

request for a new predictor node, (ii) **RequestSync** to receive a local model after violation, (iii) **ResolutionAnswer** to receive a resolution response from a local predictor node. In addition, it sends **CoordinatorSync** messages for all predictors after creating a new global prediction model, or **RequestResolution** to a ask the local predictors for their prediction models.

```
StreamExecutionEnvironment env = [...];
// Set a default parallelism for all operators
env.setParallelism(1);
// Read messages from local predictors
DataStream<TopicMessage> messagesStream =
     readKafkaStream(env,
     "LocalToCoordinatorTopicId");
// Initialize the coordinator node
CompunctionEfficientCoordinator coordinatorNode =
     new CompunctionEfficientCoordinator(configs);
// Ingest the messages stream by the coordinator
DataStream<CoordinatorMessage>
     coordinatorMessagesStream =
     messageStream.map(coordinatorNode);
// Send the messages form the coordinator to the
     local predictors
writeKafkaStream(coordinatorMessagesStream,
     CoordinatorToLocalTopicId);
```

Listing 2: The coordinator Flink program.

5 EMPIRICAL EVALUATION

In this section, we evaluate our proposed system by analyzing the predictive performance and communication complexity using a real-world event streams provided by the datAcron project in the context of maritime monitoring. The used event streams describe critical points (i.e., synopses) of moving vessels trajectories, which are derived from raw AIS messages as described in [**?**]. In particular, for our evaluation experiments we used a data set of synopses that contains $4,684,444$ critical points of 5055 vessels sailing in the Atlantic Ocean during the period from 1 October 2015 to 31 March 2016. We used it to generate a simulated stream of event tuples i.e., *(id,timestamp, longitude, latitude, annotation, speed, heading)*, which are processed by the system to attach an extra attribute *type* that represents the event value, where $type \in \Sigma$, and $\Sigma = \Sigma_1 = \{$ *VerySlow ,Slow, Moving, Sailing, Stopping* $\}$, which is based on a discretization of the speed values. Or $\Sigma = \Sigma_2 = \{$ *stopStart,stopEnd, changeInSpeedStart, changeInSpeedEnd, slowMotionStart, slowMotionEnd, gapStart, gapEnd, changeInHeading* $\}$, which is derived based on the values of the *annotation* attribute that encodes the extracted trajectory movement events [**?**]. In our experiments, we monitor a pattern $\mathcal{P}_1 = Sailing$ with Σ_1 that detects when the vessel under way (sailing). Likewise, we test a second pattern $\mathcal{P}_2 =$*changeInHeading; gapStart; gapEnd; changeInHeading* with Σ_2.

Experimental setup. We ran our experiments on single-node standalone Flink cluster deployed on an Ubuntu Server 17.04 with Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz X 8 processors and 32GB RAM. We used Apache Flink v1.3.2 and Apache Kafka v0.10.2.1 for our tests.

Evaluation criteria. Our goal is to evaluate our distributed pattern prediction system, which enables the synchronization of prediction models (i.e., PMC models) on the distributed predictor nodes. Our proposed system can operate in three different modes of synchronization protocols/schemes: (i) static scheme based on synchronizing the prediction models periodically every b of input events in each stream, (ii) continuous, full synchronization for each incoming event (hypothetical), (iii) dynamic synchronization protocol based on making the predictors communicate their local prediction models periodically but only under condition that the divergence of the local models form a reference model exceeds a variance threshold Δ (recommended). We compare our proposed system against the isolated prediction mode, in which models are computed on single streams only, and compare the predictive performance in terms of : (i) $precision = \frac{\#\ of\ correct\ predictions}{\#\ of\ total\ predictions}$ is the fraction of the produced predictions that are correct (i.e., a full match occurred within the prediction interval). (ii) $spread = end(I) - start(I)$ is the width of the prediction interval I.

Moreover, we study the communication cost by measuring the *cumulative communication* that captures the number of messages that are required to perform the distributed online learning modes to synchronize the prediction models. Next, we present the experimental results for the patterns $\mathcal{P}_1 = Sailing$ with an order of $m = 2$, and $\mathcal{P}_2 =$*changeInHeading; gapStart; gapEnd; changeInHeading* with first order $m = 1$. All experiments are performed with setting the batch size to 100 ($b = 100$), the variance threshold of 2 ($\Delta = 2$), 80% as PMC prediction threshold ($\theta_{fc} = 80\%$), and 200 for the maximum spread.

Experimental results. Figure 4 depicts the average precision scores of predictions models (one prediction model per vessel) of all synchronization modes for the first pattern $\mathcal{P}_1 = Sailing$, namely, isolated without synchronization, continuous (full-sync), static, and our recommended approach based on the dynamic synchronization scheme. It can be clearly seen that all methods of distributed learning outperform the isolated prediction models. The hypothetical method of full continuous synchronization has the highest precision rates, while the static and dynamic synchronization schemes have close precision scores. Consequently, dynamic synchronization is not much weaker than the static synchronization, but requires much less communication, as explained below.

Figure 5 provides the amount of the accumulated communication that is required by the three modes of the distributed online learning, while, the isolated approach

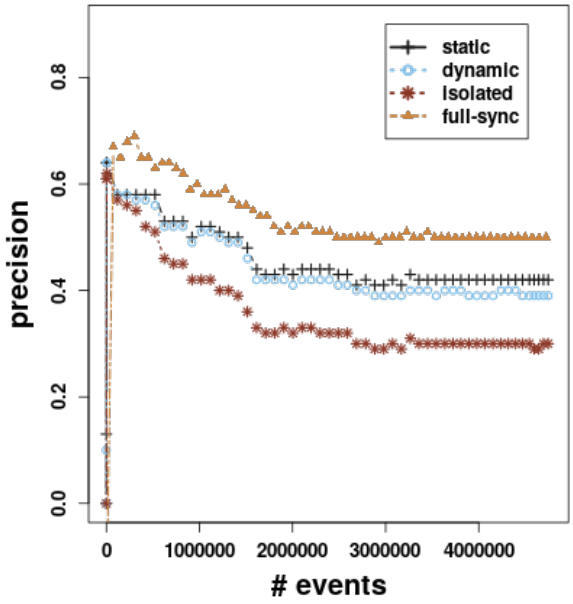


Figure 4: Precision scores with respect to the number of input events over time for $\mathcal{P}_1$.

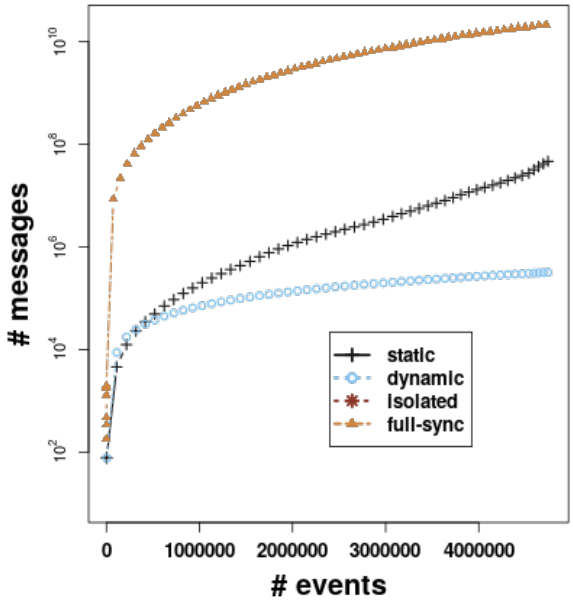


Figure 5: Commutative communication with respect to the number of input events over time for $\mathcal{P}_1$.

does not require any communication between the predictors. These results are shown for $\mathcal{P}_1$. As expected, a larger amount of communication is required for the continuous synchronization comparing to the static and dynamic approaches, and it can bee seen that we can reduce the communication overhead by applying the dynamic synchronization protocol (a reduction by a factor of 100) comparing to the static synchronization scheme, even with a small variance threshold $\Delta = 2$. Furthermore, the dynamic protocol is still preserving a close predictive performance to the static one (see Figure 4). Therefore, we will only consider the dynamic synchronization and the isolated approach in the evaluation of the second pattern.

In Figure 4, we also note that the precision is going down in a first phase and stabilises then. This seems to be counter-intuitive, as the models should improve when getting more data up to a certain point. For explanation, we have investigated the effect of the distributed synchronization of the prediction models on the average spread value, Figure 6 shows the spread results for all approaches. It can be seen that the spread is higher for the distributed learning based methods comparing to the isolated approach. Furthermore, the average spread is decreasing over time until convergence, as result of confidence increase in the models. This may explain the drop in the precision scores from the beginning until reaching the convergence. We will investigate further in the interrelation between precision and spread in future work.

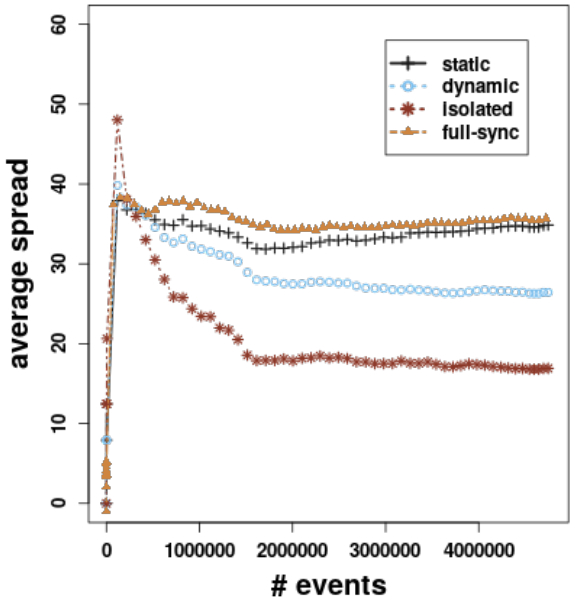


Figure 6: Average spread for $\mathcal{P}_1$.

For the second, more complex pattern ($\mathcal{P}_2$), we have found that the precision was worse for a distributed model generated over all vessels than in the model created for each vessel in isolation. This indicates that there is no global model describing the behaviour of all models consistently. However, when looking at specific groups of vessels, we achieved an improvement in terms of precision. As initial

experiment, we only enable the synchronization of the prediction models associated with vessels that belong to the same type. Currently, this change is technically performed by an extra filter step that passes only one type of vessels at time, while multiple runs of the system are required for all vessel types. Fore example, Figure 7 shows the precision scores for vessels of type *PLEASURE CRAFT*. An interesting observation is that the dynamic synchronization approach still has a higher precision scores than the isolated approach. We will further investigate the effect of groupings and further patterns in future work.

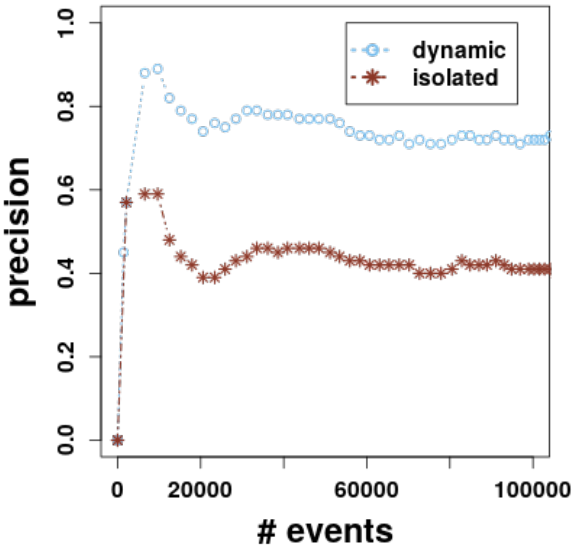


Figure 7: Precision scores of $\mathcal{P}_2$ for *PLEASURE CRAFT* vessels.

6 CONCLUSION

In this paper, we have presented a system that provides a distributed pattern prediction over multiple large-scale event streams of moving objects(vessels). The system uses the event forecasting with Pattern Markov Chain (PMC) [**?**] as base prediction model on each event stream, and it applies the protocol for distributed online prediction [**?**] protocol to exchange information between the prediction models over multiple input event streams. Our proposed system has been implemented using Apache Flink and Apache Kafka, and empirically tested against a large real-world event streams related to trajectories of moving vessels.

ACKNOWLEDGMENTS

This work was supported by the EU H2020 datAcron project (grant agreement No 6875).