Sage: A Multiuser Cooperative Controller for Mobile Edge Systems

Nuno Coelho, Diogo Ribeiro, and Hervé Paulino(✉)

NOVA Laboratory for Computer Science and Informatics, Computer Science Department, NOVA School of Science and Technology, NOVA University Lisbon, Caparica, Portugal
{ng.coelho,dpi.ribeiro}@campus.fct.unl.pt, herve.paulino@fct.unl.pt

Abstract. Mobile devices have become ubiquitous, being used to perform all kinds of daily tasks, from surfing the web, reading e-mails, playing video games, reading and writing documents, and so on. As a result, the edges of the Internet have become resource-rich spaces with millions of devices. To contribute to the world of crowd-sourcing applications being developed for the edge, we propose a new category of applications that require participating users to cooperate among themselves, in order to achieve a common goal. To this end we propose Sage, a distributed middleware for the real-time aggregation of events in the specific context of cooperative control. Sage is a generic framework able to aggregate data received, from a small number to thousands of sources, into a stream of events to be handed out to the final application.

Keywords: Edge Computing · Mobile Computing · Cooperative Control · Real-Time Data Aggregation

1 Introduction

The massive concentration of devices at edge networks spawn rich environments in both human and machine resources. These new environments can be leveraged to create novel applications that require collaboration among the edge devices. Such collaboration can be achieved by creating crowd-sourced computing opportunities that require a human element, and also by solving the power and energy limitations inherent to mobile devices [7].

Several studies have already examined various social and technical aspects of user collaboration over an edge network. In [19], Vajk et al. demonstrated that the inclusion of sensors in mobile devices opened up the possibility to use them as controllers with a wide array of input possibilities. In [10], J. Leikas et al. studied the social behavior of users when interacting with a multi-user application, and perceived that the resulting social interactions are a strong motivation for using such applications.

This work was supported by FCT-MCTES via project DeDuCe (PTDC/CCI-COM/32166/2017) and NOVA LINCS (UIDB/04516/2020). We also wish to thank Samsung for supporting this work with state-of-the-art hardware.

S. Longfei and P. Bodhi (Eds.): MobiQuitous 2022, LNICST 492, pp. 313–333, 2023.
https://doi.org/10.1007/978-3-031-34776-4_17

In this context, we envision a new category of **collaborative control** applications that allow a large number of users in an edge network to cooperate towards a common objective. Users can form one or more groups and use their potentially mobile devices to participate in the application collectively.

Motivational Example. There are many application scenarios for cooperative control, from controlling characters or objects in computer games, to controlling real-world objects, such as stage lightning in a concert, to voting, among others. A real scenario builds from an activity that has been offered in past editions of our faculty's open day. It is an adaptation of the famous *Pong* [4] game that already makes use of cooperative control. The players are divided into two groups (placed in neighboring rooms) and cooperate with their group's members to control one of the game's paddles by showing or hiding green cardboards.

In our take of this adaptation, the players use their mobile phones to cooperatively control a paddle, being even able to sit in the same room.

Proposal. We propose Sage, a distributed middleware to be deployed at edge networks with the task of continuously receiving and aggregating tenths to thousands of event flows into a single one, to be fed to a final application. With the goal of accommodating a large number of event sources, a Sage deployment may comprise multiple aggregating servers that cooperate to compute the final event flow. Moreover, Sage was thought for mobile environments, addressing issues such as dynamic discovery and connection, efficient network communication, and tolerance to churn. Lastly, Sage is also a generic framework that enables programmers to tailor the system to the needs of their application by configuring several parameters, such as data rate, aggregation functions and action granularity.

To the best of our knowledge, there are several frameworks and protocols that perform data streaming and data aggregation in cloud and edge environments, but no previous work has addressed the specific challenges raised by the use of many sources to cooperatively control an asset. Therefore, the contributions of this paper are: 1. Sage, a novel distributed system for the real-time aggregation of events in the specific context of cooperative control (Sect. 2). 2. The application of Sage to the context of cooperative gaming, with two games that require the users to collaborate to achieve the games' goals (Sect. 3). 3. A comprehensive experimental evaluation that addresses issues such as scalability, adaptability, tolerance to churn and energy consumption (Sect. 4).

2 Sage

Sage is a distributed middleware that provides a generic framework for cooperative control. It enables users to cooperate among them by using their (potentially mobile) devices to perform collective operations. At its core, Sage is a system for the real-time distributed aggregation of events directed to the specific context

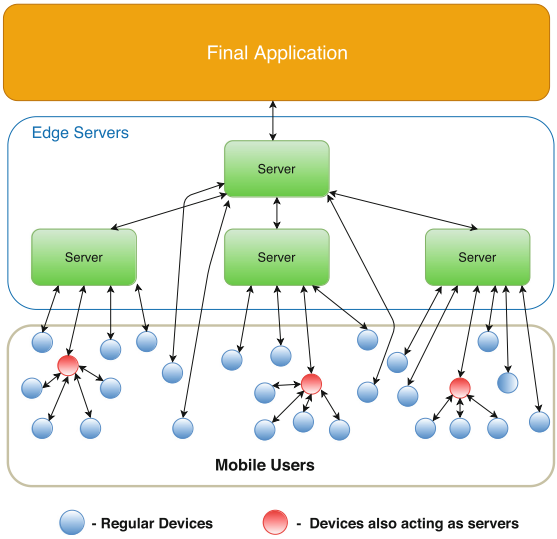


Fig. 1. System Architecture

of cooperative control. It collects and aggregates the event streams generated by the devices into a single stream to be handed out to the final application.

As displayed in Fig. 1, SAGE's architecture comprises three actors, physically distributed in an edge network: user devices, henceforth called **sources**, **servers** deployed in edge nodes connected by a LAN network and performing aggregations over incoming data, and the final **application** that receives the stream of aggregated results.

A source generates a flow of events representing actions performed by a user, such as pressing a button, choosing an option, tilting the device or moving it around. Users are assigned to groups to represent teams in a game, roles in multi-asset controlling system, and so and so forth. Source nodes have no mobility restrictions, meaning that they can move around and connect or disconnect from the system whenever they choose to. They are also independent from each other, not interacting directly among them, and thus not requiring any coordination.

The servers have the purpose of aggregating multiple streams of incoming events into a single one. If there is a need to accommodate a vast number of sources, several servers may be deployed, spawning an event aggregation tree. The sources become the leaves of this tree, being free to connect to any server. Obeying the tree structure, every node must connect to one and only one parent and the root server must connect to the application. Note that sources may connect to any server in the tree, not only to the leaves. Restricting the connection of the sources to leaf servers would result in an unbalanced system, in which mid-level servers would only process a small number of streams from their children servers, while the leaves would have to process the data generated by all sources. When necessary, user devices may also assume the role of a server, providing for the deployment of more flexible *ad hoc* structures.

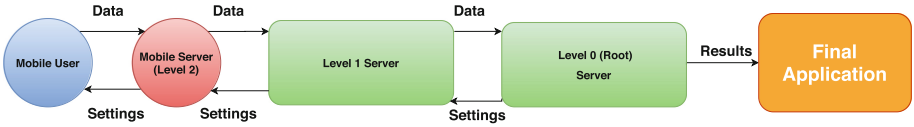


Fig. 2. Data flow between system components

Sage was designed and implemented with the rationale that sources and servers may be agnostic to the target application. The servers are thus generic computing nodes that receive streams of events and apply the aggregation functions defined in the system's deployment, while the sources may be, for instance, generic game controlling apps. Accordingly, when a new node (source or server) first connects to the system it receives a *configuration settings* message with the application's configuration parameters. Moreover, throughout their execution, nodes may adapt to changes in the overall execution context. These adaption decisions are made by servers and transmitted to their sub-tree. Figure 2 illustrates the flow of action events and of settings messages. We will discuss both in the following sections.

2.1 Action Events and Configuration Settings

Sources generate events that represent an action performed by a user. These events always flow upwards in the Sage tree and report a value bound to a particular action type performed by a particular group, *e.g.*: *moving a game paddle with vector value v by an element of group g*. To that end, an action event comprises several attributes that we now detail:

Source id - application-wide unique identifier that identifies the action's source node. Depending on the context, it may be either a Universally Unique IDentifier (UUID) or a uniformly distributed 32 bit random number.
Source type - may be a user device or a non-root server.
Group id - integer internally generated to represent an application-given group name. Only the group name is made known to the users (for instance, for them to choose the team they want to play on) but all communication makes use of the associated id.
Event id - application-wide event identifier.
Event value - value communicated in the event.
Event weight - number of sources that cooperated to produce the event's value: 1, in the case of a source; the number of sources that contributed to the aggregated value of the event, in the case of a server.
Group size - optional value that indicates the size of the group at the time of the action. It is useful for programming event-processing functions that take into account the size of the source population that could have contributed to the action versus the size of the one that actually did (the action's weight). Looking at the Pong motivational example (Sect. 1), moving to the left or to

the right will depend on the relative weight of such events, as well as on the percentage of sources that actually moved. If most did not (neither left nor right), the decision has to be *not to move*.

Actions may be of two kinds: *discrete* or *continuous*. As its designation implies, the former classifies actions that take values in a discrete domain, such as pressing a button, while the latter classifies actions that take values in a continuous domain, such as tilting a device.

Due to their nature, continuous actions persist and evolve in time. To avoid having useless information repeatedly sent to the servers when there is no significant change in the action's value, we associate a *time-to-live* (TTL) and a *granularity* threshold to each action of this kind. The TTL is used to avoid sending the same event over and over, when a user is performing the same action for a while. A continuous action event is hence assumed to be valid in the system until its TTL expires or a new event of the same action type is generated by the same source. On the source's end, as soon as a different action of the same type is performed, a new event is issued. On the other hand, if the original action persists and the TTL expires, the event is resent and the TTL renewed. Besides greatly reducing network traffic, this approach also provides means for masking temporary disconnections by source nodes.

Regarding granularity, a change in an action's value, although yielding a value different from the previously reported, may not be relevant enough to generate a new event. To avoid generating events that are not distinct enough from the last one sent, we divide the action's domain in chunks of a given size (the *granularity*) and only consider as new values the ones that cross a chunk's boundary. An immediate application case is the use of a device's tilt to perform a paddle movement. The change in the device's inclination may not be relevant enough to generate a new action value.

As introduced in the beginning of Sect. 2, when a new node connects to the system it receives an *application configuration settings* message that includes the *parent*'s identifier; the description of the *actions* allowed, with id, type (discrete or continuous), TTL and granularity; the name and id of the *groups* to choose from (if allowed) or the *source's group*, if one was automatically assigned; the *transmission settings* that convey the application's requirements regarding the stream's rhythm (given as the period in milliseconds between events), and; the maximum number of clients allowed per server (if defined).

2.2 Action Event Processing

A node periodically transmits events to its parent, respecting the period defined in the settings message. For example, if the application specifies a transmission period of 200 ms, the root server will send aggregated action events to the application every 200 ms.

The number of events included in the aggregation depends on the number of groups, and on the number of actions defined per group. On the stream sent by the root server to the application, an aggregation comprises values for all

continuous actions plus values for any discrete action that has been triggered during the interval. Continuing with the example above, if the application defines two teams and two actions per team, one continuous and one discrete, the root server produces a stream of aggregated events with a cadence of 200 ms. Each event aggregation must include a minimum of 2 events (the continuous actions) and may go up to 4 (the number of actions). Between the remaining nodes, the payload is reduced with the use of the aforementioned TTL and granularity mechanisms.

The nature and/or complexity of the aggregation algorithms, the volume of the incoming data (which may become too big for the server to process and aggregate in time) or network congestion (which may prevent the server from receiving enough data to process) may precipitate scenarios on which a server has no data to send when a transmission period expires. To ensure the desired output rate, SAGE adapts the transmission period of servers and sources to the conditions of the current execution context, namely to the volume of events produced and the ability of the servers to process them in time. When a server detects that it cannot keep the pace or it can actually process more data, it uses a *congestion control algorithm* to compute a new transmission period that it transmits to its children (more details in Sect. 2.6). The root server is the only node that does not receive adaptation messages, so its transmission period never changes. All the remainder nodes adapt their transmission settings whenever they are requested by their parents, and perform the task independently.

The adaptation process only considers continuous actions because discrete ones cannot be measured in a continuous time or space interval. It is, thus, not possible to predict when an event of such type is going to be issued.

2.3 Source Nodes

The architecture of a source node comprises several modules, of which we highlight four. The NetworkLayer is responsible for handling all communications with the server, sending the events generated by the node and receiving (and routing to the SystemController) all settings messages. The SensorLayer listens to all the configured device sensors, pre-processing the captured data and subsequently passing it to the SystemController for further processing. The UserInterface shows the application's information and detects all button pressings, which are also communicated to the SystemController. Lastly, the SystemController is the source's decision center. It processes the settings messages to adjust the node's configuration, namely the period between consecutive events, and processes the actions reported by the SensorLayer and the UserInterface to generate action events (respecting the actions' and the transmission settings).

2.4 Server Nodes

A server collects and processes the incoming streams and produces a single one that it transmits to its parent in the tree or to the application. Additionally, it also generates and sends adaptation requests to its children.

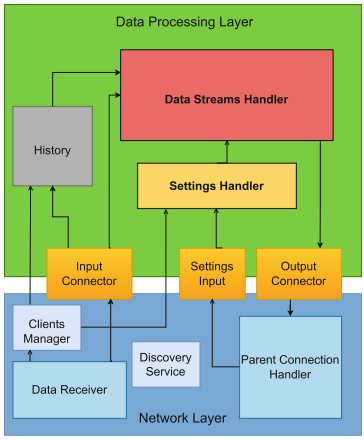


Fig. 3. Server architecture.

A server's architecture (displayed in Fig. 3) is composed of two layers: the Network Layer and the Data Processing Layer. In this paper we will briefly present the former and focus mainly on the latter.

Network Layer. The Network Layer is responsible for managing connections with clients and other servers, as well as receiving and sending data. It comprises four primary components: the Clients Manager manages the connectivity and communication with the server's children, the Data Receiver receives the action event streams, the Parent Connection Handler handles all communication with server's parent and the Discovery Service responds to server discovery requests.

Data Processing Layer. The core component of the Data Processing Layer is the Data Streams Handler, which makes use of a 6-stage Edgent [1] streaming topology to filter and aggregate the event streams. The Source stage simply routes the data received from the Network Layer to the following stages. The Group Split and the Action Split stages split a single input stream into multiple output streams. The first does it to produce a stream per group defined in the application's configuration settings and, the second, to produce a stream per action within a group (also for each action defined in the application's configuration settings). At the end of these 3 initial stages we have one stream per action per group. The Aggregation stage applies the aggregation function (a Java functor) defined for each pair (action, group), being the results merged into a single stream by the Union stage and routed to the Network Layer by the final Sink stage. Figure 4 illustrates the topology built to process streams in a scenario comprising two groups, 'A' and 'B', and two actions per each group, 'X' and 'Y'.

The input streams are discretized into batches that contain all the event values received (or collected from History) in the time window of the last out-

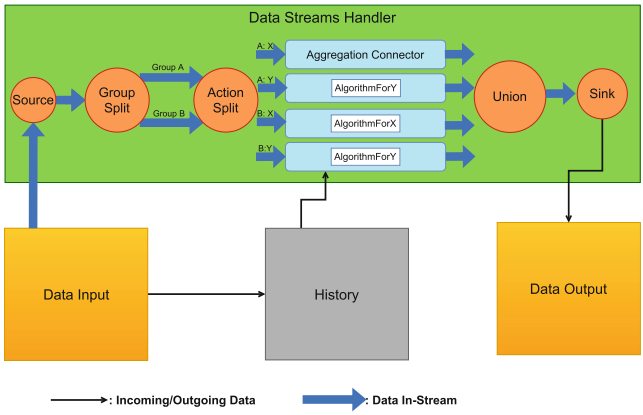


Fig. 4. Data Streams Handler streaming topology

going transmission period. The aggregation function is defined by extending the `Algorithm` abstract class and implementing its method `apply`:

```
abstract class Algorithm<In, Out> {
  abstract Pair<Out, Integer> apply(Collection<Pair<In, Integer>> data);
}
```

The method receives a collection of pairs, whose first element represents the value of the event, of type `In`, and the second element represents the event's weight. The result is a single pair of an output event type `Out` and its weight. If the return value is `null`, the framework assumes that no event is to be issued.

The class features other methods, such as `emit(Pair<T, Integer> data)`, `getPopulation()` and `oneValuePerSource(boolean value)`. The first enables the algorithm to generate more events than the one that is returned. The second returns the number of sources (of the given group) that are directly or indirectly connected to the server. As explained in Sect. 2.1, this information is elemental to develop algorithms that take the population's size (e.g. the number of players) in consideration when computing an aggregation. The third informs the framework to include all or just a single value per source in the data collection handed to the algorithm's `apply` method. The default is *multiple values*. If the *one value* option is chosen, only the most recently received value is kept.

Handling Continuous Actions. As introduced in Sect. 2.1, the value of a continuous action has a TTL that enables the server to deduce the action's current value from previously recorded data and, with that, reduce network traffic and mask temporary disconnections. For that, it makes use of the History module that stores the last events received per client for every continuous action (until the action's TTL expires). It thus keeps a relation between client identifiers, action identifiers and the last value received for each (client, action) entry.

When a child fails to transmit data for a continuous action in the interval defined by the sending period, the Aggregation stage detects the lack of data

and, before applying the aggregation function, contacts the History module to retrieve it. History verifies if it contains a valid event (i.e. with a not expired TTL) to match the request. If so, it relays the event to the aggregation connector, otherwise it simply discards the event (if any).

Client Management. The History module also executes a periodic background task that removes any event that is no longer valid. To that end, the task collects the nodes that have had their events expired and asks the **Clients' Manager** module to verify if such nodes are still *alive* and, if not, remove them from the current source population.

2.5 Dynamic Server Discovery

For a node (source or server) to connect to a running Sage deployment, it must find an existing server. For that purpose, the requesting node initially sends a discovery message to a predefined multicast group and waits for replies for a given time period t, with a read timeout of s seconds, with t and s configurable. The timeout assures that the servers that take more than s seconds to respond are not considered as candidates for connection.

Each server responds with its IP address and the number of connected clients, enabling the requesting node to choose the least occupied option. Currently, we are not considering processing power as a metric for server selection. We intend to address this issue in future work.

To dynamically balance the system, non-server nodes periodically run the discovery process to check if better options are currently available. If so, the node waits for a random period of time $\leq 5\,\text{s}$, and checks again. Whenever the conditions persist, the node simply disconnects from the current server and connects to the new one. Otherwise, if there is still a better server than the current but it is not the same as the one found before, the waiting process is restarted. The waiting period serves to avoid massive migrations when new servers arrive or reconnect to the system after temporary absence.

2.6 Adaptation

A Sage deployment is expected to produce a stream of data with a regular rate, whose period is defined at deployment time. To meet this requirement, Sage servers may disseminate configuration settings message to their sub-tree, causing their children to adjust their execution parameters and, if needed, disseminate a settings message of their own.

In a server, every settings message (either received or generated) is routed to a second Edgent topology to demand the Parent Connection Handler to change its sending period and/or to adapt the period of its sub-tree. To allow the system to stabilize, adaptation decisions may only be made every S seconds, with S being a value defined in the deployment settings (defaulted to 3).

A Sage server is parameterizable w.r.t the period adaptation algorithm, which must implement the `PeriodAdapter` interface that defines two methods:

```
interface PeriodAdapter {
  int increase(int... lastPeriods);
  int decrease(int... lastPeriods);
}
```

The `increase` method is called whenever the network layer is unable to retrieve a value, for each continuous action defined per group, from the data processing layer (*i.e.* from the output connector of Fig. 4). Inversely, the `decrease` method is called whenever: (1) the transmission period is longer than the one defined by the application and (2) the network layer receives values for all continuous actions of all groups, n times in a row (for a given n value defined in the deployment settings, defaulted to 5). Both methods receive the last p values computed for the transmission period (with p also defined in the deployment settings, defaulted to 1) and must return the new value for the period.

Starting from a configured transmission period p, an adapted period may only take values in interval $[\frac{3}{4} * p, 10 * p]$. Values smaller that the lower limit would force children nodes to send more values than required, ultimately wasting resources and compromising data fidelity. The upper limit prevents continuously increasing the period in abnormal scenarios, such as temporary network unavailability.

The Algorithms. We have implemented some algorithms based on techniques widely used in network congestion protocols, such as Additive Increase/Additive Decrease (AIAD), Additive Increase/Multiplicative Decrease (AIMD) and Cubic Increase/Multiplicative Decrease (CIMD) [9].

We also experimented with techniques used for the online adaptation of parallelism in parallel computing systems, namely we developed an algorithm inspired by *RUBIC* [12], which was originally designed for adjusting the parallelism level of transactional memory applications. *RUBIC* uses a combination of AIAD and CIMD. The frequency increase (period decrease) function begins with a cubic increase phase to quickly respond to change, proceeding then with subsequent linear increments, if necessary. The cubic function is defined as follows:

$$cubic(p) = \begin{cases} p + x \ , x < 0 \\ p - x \ , x \geq 0 \end{cases} \text{ for } x = \beta \times (\sqrt[3]{p * (\alpha/\beta)})^3$$

where p is the period before the adaptation process; α is the constant multiplication factor and β is a constant scaling factor that controls the period's reduction rate. Conversely, frequency decrease (period increase) begins softly with linear decreases and, if these persist, switches to the multiplicative factor (α).

3 Use Case: Cooperative Games

We developed an Android app for a game controller that allows the user to choose a team (if such is allowed), tilt its device left or right to move to the pretended direction and press buttons to emit actions, such as shooting or picking power-ups. The app is a SAGE source node that connects to a SAGE server and

```
ApplicationInfo pongInfo = ApplicationInfo.Builder.newBuilder()
 .setAppName("Pong")
 .setTransmissionPeriod(200)
 .addGroups("Racket 1", "Racket 2")
 .addToAllGroups(new ContinuousAction("Horizontal Movement", new MovementAlgorithm(), 350))
 .build();
Sage.newServer(pongInfo).start();
```

Listing 1: The application information of Cooperative Pong

```
public Pair<Direction, Integer> apply(Collection<Pair<Direction, Integer>> data) {
    int left = 0, right = 0, mid = 0;
    for (var pair : data)
        switch (pair.getKey()) {
            case LEFT -> left += pair.getValue();
            case MIDDLE ->  mid += pair.getValue();
            case RIGHT -> right += pair.getValue();
        }
    if (right == left || (mid >= left && mid>= right)) return Pair.of(MIDDLE, mid);
    else if (right >= (left + mid)) return Pair.of(RIGHT, right);
    else return Pair.of(LEFT, left);
}
```

Listing 2: Movement algorithm

generates a stream of actions events. The tilting is detected by adding the *game rotation vector* sensor [8] to the node's Sensor Layer and transforming its output into tilting degrees. We used this app as a controller in several games. Here we confine our discussion to the motivational example presented in Sect. 1 and the somewhat akin Arkanoid.

Cooperative Pong. The players only need to use the tilt functionality, tilting the devices in the direction they want the paddle to move. If they do not want to move the paddle, they need only to hold their devices in a horizontal position.

Listing 1 showcases the configuration of a server. The `ApplicationInfo` class provides for the definition of the application's settings. In this case, two groups (teams `Racket 1` and `Racket 2`) and a single action (`Horizontal Movement`) were defined. The action is continuous and was added to both groups (given that both teams have the same role). The aggregation algorithm (Listing 2) is a consensus algorithm that takes a collection of votes for moving left, right or not moving at all, and returns the direction with the most number of votes (V), paired with V. For example, if the winner is *left* with 100 votes, the result will be (LEFT, 100).

Cooperative Arkanoid. The second use case is the widely known *Arkanoid*, initially developed by the Taito Corporation [2]. This game has only one paddle, but users can pick up and activate power-ups. Following those specifications, we defined two actions: the paddle movement (as in Pong) and the power-up activation. The latter is a discrete action triggered by pressing one of the mobile controller's buttons. The aggregation algorithm is a two level algorithm. At the

Table 1. Specifications of the smartphones used in the test scenarios.

Device	Xiaomi Mi A1	Samsung S10+
CPU	Cortex-A53 Qualcomm Snapdragon 625 8×2GHz	Qualcomm SDM855 Snapdragon 855 1×2.84 GHz Kryo 485 & 3×2.41 GHz Kryo 485 & 4×1.78 GHz Kryo 485
RAM	4 GB	8 GB
Battery	3080 mAh	4100 mAh
Wi-Fi	WiFi (802.11 a/b/g/n/ac)	WiFi (802.11 a/b/g/n/ac)
OS	Android 10	Android 12

mid-level servers it returns the sum of all received values (discarding duplicates), and the corresponding weight. At the root server, it compares the weight of the action with the source population to check if, at least, half of the population pressed the button. If so, the action is relayed to the app, if not, it is discarded (the algorithm returns `null`).

4 Evaluation

In this section we evaluate Sage to assert that it aggregates and delivers data efficiently under different conditions and workload assumptions. More specifically, our experiments aim to answer the following questions:

Q1. Scalability - How well does the Sage scale with the increase of the number of sources?
Q2. Adaptation - What is the impact of the different adaptation algorithms in the system's performance?
Q3. Computational Complexity - How does the system perform when using aggregation algorithms with a significant processing overhead?
Q4. Churn - How well does a server handle the connection and disconnection of a large number of clients, while processing data?
Q5. Smartphones as Servers - How does the mobile device perform as a server?
Q6. Energy Consumption - How much energy is consumed by the mobile application when used as a simple client and as a mobile server?
Q7. Granularity - What is the impact of the defined granularity in a client's output rate?

Test Environments. We evaluated the behavior and performance of Sage in two environments: **Real** - composed of physical source devices connected to a single stationary server. We used Android mobile devices (Table 1) running the controller app described in Sect. 3 and connected to the server via WiFi. **Emulated** - composed of one or more Sage source node emulators, each running multiple nodes. We have used this emulation environment to evaluate other works

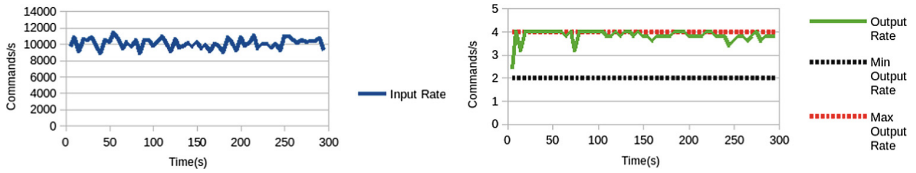


Fig. 5. Scalability: (servers: 1, sources: 7000, period: 1000)

such as [6,16,17]. In this particular setting, each emulator and server runs in a dedicated physical node of a computing cluster.

The *real* environment allowed us to evaluate and assert Sage's usability in real-life applications and answer questions Q5 to Q7. The *emulated* environment allowed us to scale the number of sources to numbers that we could not do with the *real* test-bed and, with that, address questions Q1 to Q4.

Experiment Configuration. An experiment configuration in the emulated environment is specified by multiple parameters, namely: the number of servers, the number of sources, the transmission period and the adaptation algorithm. All experiments use a deployment configuration that features two teams, each one able to perform a continuous and a discrete action. Furthermore, all experiments had a duration of five minutes, with metrics being recorded every second.

Along this section we will use the term **failure** to designate an occasion on which the output rate does not meet the requested transmission period.

4.1 Scalability

Our first set of experiments aims to discover on how many clients a server can accommodate, while meeting the transmission period requested by the application. With the goal of stressing the system as much as possible, we disabled the adaptation process for these experiments. So, the base configuration for these experiments is (servers: 1, adaptation: none), with the number of sources and the transmission period varying among experiments.

For the first experiment, we set a transmission period of 1 s and the number of sources took values 1000, 2000, 4000, 5000, 6000 and 7000. The input and output rate of the server in all scenarios showed little variation, with the server comfortably dealing with the load. Figure 5 showcases the results for the 7000 client scenario.

To further stress the system, we reduced the period to 200 ms, *i.e.* a minimum of 10 events every second, and fluctuations in the input and output rates started to appear with more than 2000 sources. Although the large amount of input events (12 000 to 16 000 per second), the output rate still managed to stay relatively stable, only varying between 13 and 15 events per second. However, this was the first scenario on which we observed failures, which lasted for about 3 s. These failures became more frequent as the number of sources increased.

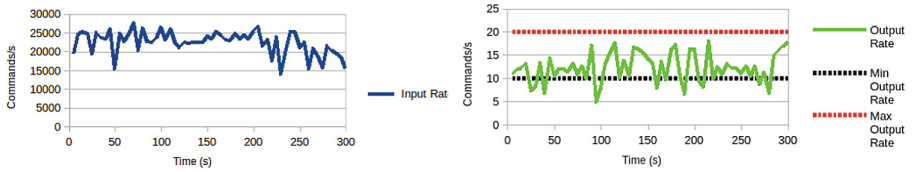


Fig. 6. Scalability: (servers: 1, sources: 5500, period: 200)

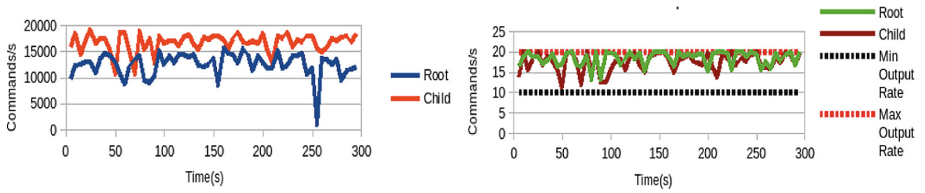


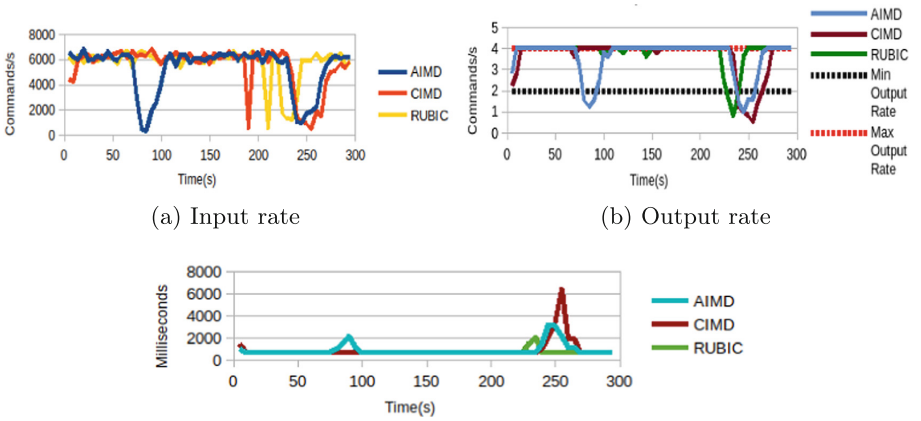
Fig. 7. Scalability: (servers: 2, sources: 5500, period: 200)

Figure 6 depicts the results for 5 500 clients, where it is visible that the server fails to meet the transmission period requirement.

The solution for such situations is to distribute the load among multiple servers. To assess Sage's horizontal scaling, we repeated the last scenario with 2 servers; the results are presented in Fig. 7. Although some occasional variations in the input and output rates, the improvement of the system's performance is visible, with the output rate never falling below the desired minimum of 10 events per second. We further experimented with a transmission period of 50 ms only to confirm our findings. In these experiments the server starts to struggle with 4000 sources. So, as expected, the smaller the period, the fewer sources the servers can accommodate without compromising the expected output rate. Also, the use of more servers effectively shares the load. The dynamic server discovery and load balancing algorithm (Sect. 2.5) works very well in homogeneous networks, allowing the system to scale if $\frac{c}{S} \leq c_{limit}$, where c is the number of sources, s is the number of servers and c_{limit} is the upper limit for the number of clients per server. This limit depends on many variables, such as the number of clients and the complexity of the aggregation algorithms, but can be set in the deployment configuration. A server will not accept more connections than the limit imposed. As mentioned in Sect. 2.5, addressing heterogeneous networks is future work. Given the work in the field, it is more of a technical challenge than a research one.

4.2 Adaptation

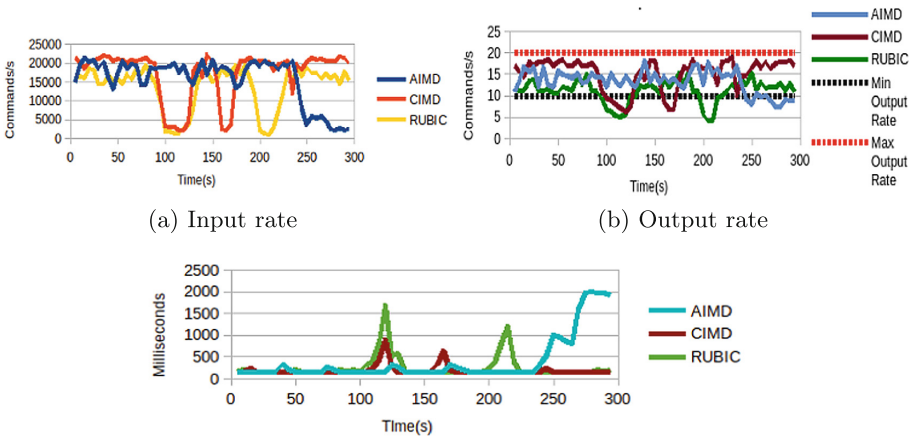
Next, we analyze the impact of Sage's period adaptation process, comparing the 3 algorithms implemented: AIMD, CIMD, and RUBIC. We are particularly interested in how the system reacts to sudden changes in the input and output



(a) Input rate (b) Output rate

(c) Transmission period

Fig. 8. Adaptation: (servers: 1, sources: 4000, period: 1000)



(a) Input rate (b) Output rate

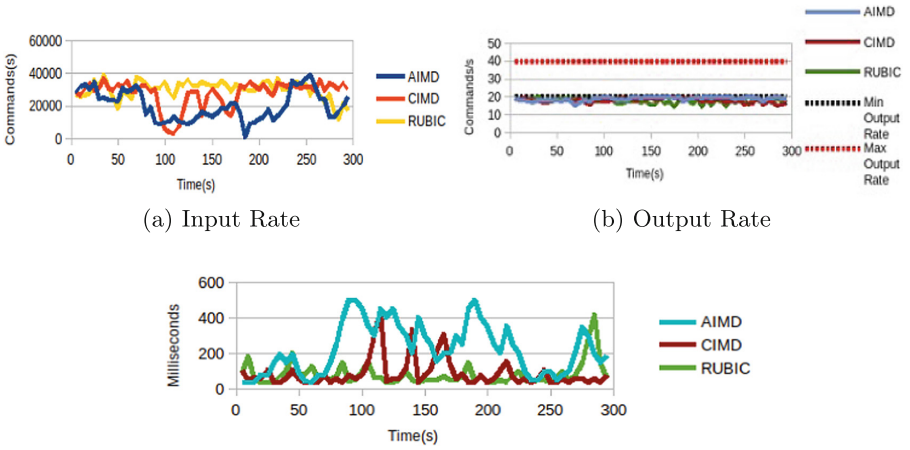
(c) Transmission period

Fig. 9. Adaptation: (servers: 1, sources: 4000, period: 200)

rates. Hence, the algorithms were parameterized with $\alpha = 0.7$, meaning that the incremental function of *CIMD* and the decremental function of *AIMD* change their interval aggressively[1], and with $\beta = 0.1$, to limit the incremental function of *CIMD* to the least possible.

For each algorithm we considered six scenarios: (servers: 1, sources: 500 or 4000, period: 1000, 200 or 50 ms). Additionally, at every second, sources have now a fixed probability of entering a pause period that may go from 1 to 4 s.

[1] The higher the α factor, the steepest is the *CIMD* increase and the *AIMD* decrease.



(a) Input Rate

(b) Output Rate

(c) Transmission Period

Fig. 10. Adaptation: (server 1, sources: 4000, period: 50)

This pause emulates periods of lag and network issues, and were used to analyze how the adaptation algorithms recover from failures.

The results obtained demonstrate that the adaptation process effectively allows the system to recover from failures in a few seconds. Overall, the evaluated algorithms control the rate at which clients send data, asking them to decrease their frequency when the server is not able to process the data in due time, but also to increase it in opposite scenarios. Figures 8, 9 and 10 depict the results for, respectively (servers: 1, sources: 4000, period: 1000, 200 and 50 ms). To the plots of the input and output rates, we add a new plot with the transmission periods resulting from the adaptation process.

No adaptation algorithm was a clear winner, but the *CIMD* and *RUBIC* have shown to be more versatile. *RUBIC* displays the least variation in the input and output rates in most experiments. *CIMD* has presented a somewhat larger variation, explained by the fact that the incremental function is cubic and the decremental function is multiplicative. Lastly, *AIMD* revealed to be an algorithm very sensitive to changes in the input and output rate, especially in scenarios with small periods and a low number of clients. In many moments the algorithm increased the interval consecutively and was not able to recover.

4.3 Computational Complexity

The next set of experiments aim to study the system's sensitivity to the computational complexity of the aggregation algorithms. To achieve this, we modified the previously used aggregation algorithms to include a loop that busy-waits for an indicated amount of time, denoted as the overhead, before the actual data processing. The base configuration is: (server: 1, sources: 500 and 4000, period: 1000, 200 and 50 ms, adaptation: RUBIC)

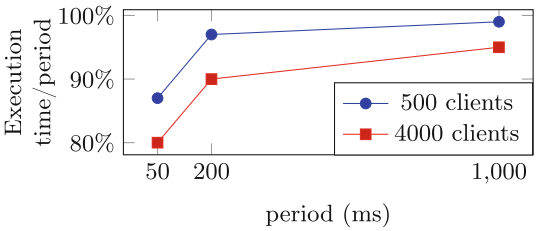
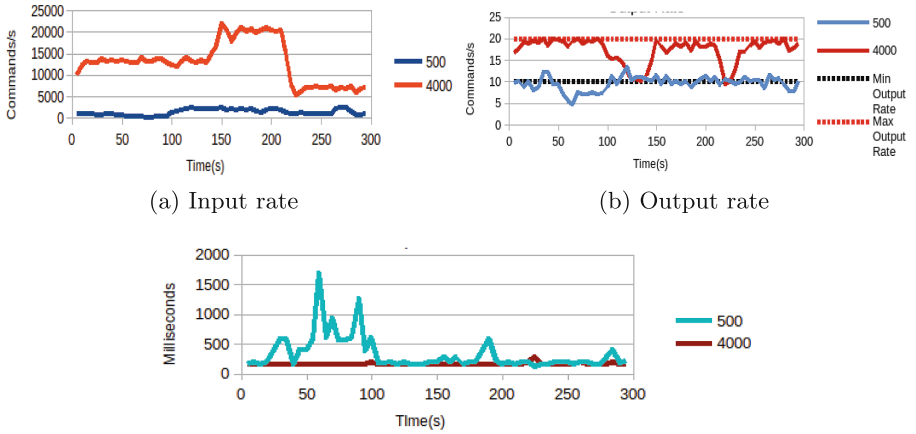


Fig. 11. Ratio between the algorithm execution time and period over which failures consistently happen.



(a) Input rate

(b) Output rate

(c) Transmission period

Fig. 12. Churn: (server: 1, sources: 500 and 4000, period: 200, adaptation: RUBIC). Half of the clients connect at 90 s and half disconnect at 210 s

For each scenario we increased the overhead until the system was not able to constantly keep the desired output rate. Figure 11 presents the results. For the lighter load of 500 clients, the server was able to keep the pace up until overheads of, respectively, 87%, 97% and 99% of the transmission period for periods of 50, 200 and 1000 ms. As expected, with more clients, the percentages decrease, in these cases to, respectively, 80%, 85% and 90%. Nonetheless, these results allow us to conclude that, in itself, the overall aggregation process is quite light, enabling the use of aggregation algorithms with execution times very close to the transmission period.

4.4 Churn

The next experiment aims to assess how a Sage server responds to abrupt changes in its number of clients. This type of scenario may happen due to network-related causes, to system-related causes, such the failure or appearing of a server, or to changes in number of sources.

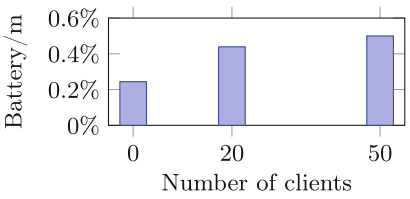


Fig. 13. Energy consumption of the mobile device, in battery % per minute

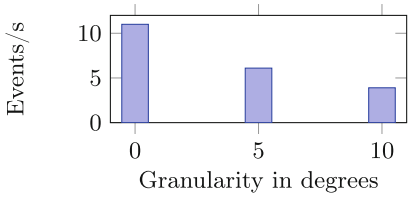


Fig. 14. Granularity: events sent with different granularity values for device tilting

Two experiments were defined: (servers: 1, sources: 500 and 4000, period: 200, adaption: RUBIC). Both begin with only the half of the total population, and have the other half connect passed 90 s. 120 s later (instant 210), half of the population disconnects. The results in Fig. 12 demonstrate that the connection and disconnection of 250 clients does not have a considerable impact in the system's performance. The connection caused a slight increase in the transmission period, but the adaptation algorithm was able to recover quickly. With 4000 clients the impact is greater. After the 90 s mark, the system took a long time to accept all connections, as depicted by the heavy drop in the output rate, but eventually managed to accept them all. At the 210 s mark, the system's performance dropped once again, but it quickly recovered and was always able to keep the minimum of 10 events per second.

4.5 Smartphones as Mobile Servers

The following experiment analyzed the performance of smartphones that work as aggregation servers, as well as standard clients. We expect mobile servers to serve tenths of clients not more. With this in mind, we set up two scenarios with the following configurations: (servers: 1, sources: 20 and 50, period: 50 ms, adaptation: RUBIC). Every source was running in the emulator and connected to the mobile server.

In both experiments, the mobile devices (of Table 1) were able to maintain a stable output rate of events. In the 20 source scenario, the servers received an average of 1108 events per second and output an average of 20 events per second. In the 50 source scenario, the device received an average of little more than 1115 events per second and output an average of 23 events per second. These results show that mobile devices can effectively act as mobile aggregation servers, whenever there is a need to reduce the load on the edge nodes or simply to have the system work in infrastructure-less environments.

4.6 Energy Consumption

We analyzed the energy consumption of a mobile device as a simple client and as a mobile server. The results for the Xiaomi Mi A1 device are displayed in

Fig. 13. We considered three scenarios: the device is used only as a source (0 clients), the device is also working as a mobile server with 20 and 50 clients.

The first conclusion to draw is that a commodity device may run our game controller app for many minutes, as the consumption per minute is around 0.2%. The addition of the server role has as greater impact, as can been observed in the 20 client bar. This slop is, however, much smaller when adding more clients. The impact comes mostly from the server engine itself than from the number of clients. To this contributes the fact that the *Arkanoid* deployment uses two simple aggregation algorithms. Computationally heavier algorithms may drain the battery much faster and should likely be entirely run by stationary servers.

4.7 Granularity

We performed tests to evaluate the impact of various granularity thresholds for the tilt of the device on the control of game characters. We are interested in the feedback of the users concerning playability and also on the number of events sent with each configuration. To that end, we invited several persons to experiment our version of the *Arkanoid* game using our game controller app. The granularity thresholds were set to $0°o$, $5°$ and $10°$.

There were significant differences in the average number of events sent per second for each threshold, as illustrated in Fig. 14, demonstrating that the mechanism has impact on network traffic and ultimately on the final results. The users reported that the $5°$ configuration was the one that provided the smoothest experience. This is the configuration that we are currently using with good game experience. However, it is essential to consider that different applications may require different thresholds, as the granularity value is inversely proportional to the number of events sent by the devices.

5 Related Work

Frameworks like Spark [21], Storm [18] and Flink [3] allow for the real-time aggregation of streamed data in cloud environments. There have also several proposals specifically designed for (or adapted to) edge and/or IoT environments. Prominent examples are Symbiosis [13], Azure Stream Analytics [11], SpanEdge [15], Amnis [20] and others, such as [5,14].

Most of these systems solve several of the challenges that we are addressing, namely the ones related to data aggregation of streamed data, such as discretizing data streams in time windows and aggregating data according to programmer-given algorithms. Some also address dynamic server discovery. However, none address the challenges raised by cooperative control, namely the adaptation to a rate imposed by an external entity, the use of historical data to mask temporary disconnections, the use of event value similarity to avoid sending the same data repeatedly.

6 Conclusions and Future Work

In this paper we presented Sage, a novel distributed system for the real-time aggregation of events in the specific context of cooperative control. We have applied Sage to the context of cooperative gaming with very good results, as it is being used to play games as cooperative Pong and cooperative Arkanoid.

A comprehensive evaluation allows us to conclude that Sage may be used from a small number of sources to thousands. In our experiments, a server is able to withstand loads of hundredths to thousands, depending on the transmission period and on the complexity of the aggregation algorithms. Higher loads can be supported by horizontally scaling the system with more servers.

From the devices' point of view, the energy consumption of our game controlling app is quite low, allowing for the app to be used for several hours.

To work presented is novel in the field and opens opportunities for the development of new types of applications at the edge. Future work will focus mainly on such enterprise – the development of new cooperative control applications in different contexts – and on the improving of the system itself.

References

1. Apache Software Foundation Incubator: Apache Edgent Overview. https://edgent.apache.org/docs/home.html
2. ArcadeClassics.net: Arkanoid: Classic arcade game video, history and game play overview. https://arcadeclassics.net/80s-game-videos/arkanoid/
3. Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., Tzoumas, K.: Apache flink: stream and batch processing in a single engine. Bull. IEEE Comput. Soc. Tech. Committee Data Eng. **36**(4) (2015)
4. Centre for Computing History: Atari Pong. http://www.computinghistory.org.uk/det/4007/Atari-PONG
5. Dautov, R., Distefano, S.: Stream processing on clustered edge devices. IEEE Trans. Cloud Comput. **10**(2), 885–898 (2022)
6. Dias, J., Silva, J.A., Paulino, H.: Adaptive replica selection in mobile edge environments. In: Hara, T., Yamaguchi, H. (eds.) Mobile and Ubiquitous Systems: Computing, Networking and Services, vol. 419, pp. 243–263. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-94822-1_14
7. Fernando, N., Loke, S.W., Rahayu, W.: Computing with nearby mobile devices: a work sharing algorithm for mobile edge-clouds. IEEE Trans. Cloud Comput. **7**(2), 329–343 (2016)
8. Google: Sensor types | android open source project. https://source.android.com/devices/sensors/sensor-types#game_rotation_vector
9. Ha, S., Rhee, I., Xu, L.: CUBIC: a new TCP-friendly high-speed TCP variant. ACM SIGOPS Oper. Syst. Rev. **42**(5), 64–74 (2008)
10. Leikas, J., Stromberg, H., Ikonen, V., Suomela, R., Heinila, J.: Multi-user mobile applications and a public display: novel ways for social interaction. In: Fourth Annual IEEE International Conference on Pervasive Computing and Communications (PERCOM 2006), pp. 5-pp. IEEE (2006)
11. Microsoft: Streaming Analytics - Data Analysis in Real Time. https://azure.microsoft.com/pt-pt/services/stream-analytics/

12. Mohtasham, A., Barreto, J.P.: RUBIC: online parallelism tuning for co-located transactional memory applications. In: Scheideler, C., Gilbert, S. (eds.) Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2016, Asilomar State Beach/Pacific Grove, CA, USA, 11–13 July 2016, pp. 99–108. ACM (2016)
13. Morales, J., Rosas, E., Hidalgo, N.: Symbiosis: sharing mobile resources for stream processing. In: IEEE Symposium on Computers and Communications, ISCC 2014, Funchal, Madeira, Portugal, 23–26 June 2014, pp. 1–6. IEEE Computer Society (2014)
14. Renart, E.G., Montes, J.D., Parashar, M.: Data-driven stream processing at the edge. In: 1st IEEE International Conference on Fog and Edge Computing, ICFEC 2017, Madrid, Spain, 14–15 May 2017, pp. 31–40. IEEE Computer Society (2017)
15. Sajjad, H.P., Danniswara, K., Al-Shishtawy, A., Vlassov, V.: Spanedge: towards unifying stream processing over central and near-the-edge data centers. In: IEEE/ACM Symposium on Edge Computing, SEC 2016, Washington, DC, USA, 27–28 October 2016, pp. 168–178. IEEE Computer Society (2016)
16. Sanches, P., Silva, J.A., Teófilo, A., Paulino, H.: Data-centric distributed computing on networks of mobile devices. In: Malawski, M., Rzadca, K. (eds.) Euro-Par 2020. LNCS, vol. 12247, pp. 296–311. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-57675-2_19
17. Silva, J.A., Cerqueira, F., Paulino, H., Lourenço, J.M., Leitão, J., Preguiça, N.M.: It's about thyme: on the design and implementation of a time-aware reactive storage system for pervasive edge computing environments. Future Gener. Comput. Syst. **118**, 14–36 (2021)
18. Toshniwal, A., et al.: Storm@twitter. In: Dyreson, C.E., Li, F., Özsu, M.T. (eds.) International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, 22–27 June 2014, pp. 147–156. ACM (2014)
19. Vajk, T., Coulton, P., Bamford, W., Edwards, R.: Using a mobile phone as a "wii-like" controller for playing games on a large public display. Int. J. Comput. Games Technol. **2008** (2008)
20. Xu, J., Palanisamy, B., Wang, Q., Ludwig, H., Gopisetty, S.: Amnis: Optimized stream processing for edge computing. J. Parallel Distributed Comput. **160**, 49–64 (2022)
21. Zaharia, M., et al.: Apache spark: a unified engine for big data processing. Commun. ACM **59**(11), 56–65 (2016)