Pournaras *et al. Complex Adaptive Systems Modeling* 2013, 1:19
http://www.casmodeling.com/content/1/1/19

Complex Adaptive Systems Modeling
a SpringerOpen Journal

RESEARCH Open Access

A generic and adaptive aggregation service for large-scale decentralized networks

Evangelos Pournaras[1*], Martijn Warnier[2] and Frances MT Brazier[2]

*Correspondence:
e.pournaras@tudelft.nl
[1]Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology, Mekelweg 4, 2628 CD, Delft, Netherlands
Full list of author information is available at the end of the article

Abstract

Purpose: Aggregation functions are used in distributed environments to make system-wide information locally available in the nodes of a network. The computation of different aggregation functions, e.g., SUMMATION, AVERAGE, MAXIMUM etc., in large-scale distributed systems is challenging and crucial for a wide range of applications. This is especially the case when the input values of these functions dynamically change during system runtime. Related approaches of decentralized aggregation are function-dependent, interaction-dependent, assume static values or cannot always tolerate duplicates and continuously changing information.

Methods: This paper introduces DIAS, the Dynamic Intelligent Aggregation Service. DIAS is an agent-based middleware that addresses these issues with a holistic approach: an efficient availability of the distributed information in every node of the network that enables the simultaneous computation of almost any aggregation function. Such an abstraction initially requires a significant communication and storage cost and has a rather large overhead. These issues are resolved by introducing an implicit local representation and storage of the explicit distributed information: aggregation memberships in bloom filters.

Results: The performance impact of bloom filters in DIAS is critical for its applicability as it compensates and reduces the initial high communication and storage required for such an abstraction.

Conclusions: Experimental evaluation under various aggregation and resource-constrained settings shows that DIAS is an efficient and accurate decentralized aggregation service.

Keywords: Aggregation; Adaptation; Agent; Bloom filter; Consistency

Background

The increasing scale and decentralization of distributed systems and applications results in an information gap: Agents, with partial knowledge about a system, require the local availability of collective and summarized knowledge about the state of the whole system to perform decision-making, adapt execution of their tasks and meet global application objectives. Therefore, aggregation of information becomes a crucial requirement to acquire such collective and summarized knowledge for a wide range of distributed applications.

Centralized computation of aggregation functions is straightforward as the whole set of information is available in one location. However, centralized aggregation is not always

an option for reasons that may concern scalability or privacy. This paper focuses on the problem of decentralized aggregation of information distributed across the nodes of a network. Aggregation functions such as SUMMATION, AVERAGE, MAXIMUM, etc. are locally computed by each node of the network. The input of these functions can be arithmetic values collected from each node of the network as well. Communication, storage and processing costs are fundamental issues that challenge the design of a generic service for decentralized aggregation.

Related aggregation methodologies are function-dependent, interaction-dependent, assume static values or cannot always tolerate duplicates and continuously changing information (Ahmed et al. 2006; Haridasan and van Renesse 2008; Jelasity et al. 2005; Kashyap et al. 2006; Kempe et al. 2003; Nath et al. 2008). In contrast, this paper introduces a generic, agent-based and middleware for dynamic decentralized aggregation, DIAS, the *Dynamic Intelligent Aggregation Service*. DIAS is based on a holistic approach: availability of distributed information in every node of the network that enables simultaneous computation of almost any aggregation function. DIAS is based on the concept of *aggregation membership* to make this holistic approach possible. Aggregation memberships are aggregation information derived and abstracted from the explicit aggregation values. For example, an agent has memberships of other agents whose information is aggregated. Complementarily, an aggregate of an agent has memberships of aggregated information in other agents. This paper shows that such implicit information can be locally and efficiently stored in probabilistic data structures, the bloom filters (Bloom 1970).

A known problem of bloom filters is that of false positives (Bloom 1970). A false positive incorrectly denotes that some information is stored in a bloom filter when it is actually not. DIAS is able to detect inconsistencies such as duplicate and outdated information under the effect of false positives in bloom filters. This paper shows how detection is possible by *mutually checking the memberships* between the remote agents of DIAS without introducing additional communication. Experimental evaluation illustrates the efficiency and performance trade-offs of DIAS. High accuracy is achieved under a range of aggregation and resource-constrained settings.

This paper is outlined as follows: Section "Problem description" illustrates the problem description and related work. Section "System overview" outlines the architecture of DIAS. Section "Modeling of dynamics" introduces the model of dynamic aggregation in DIAS. Section "Dissemination and collection" illustrates the information dissemination and collection in DIAS. Section "Consistent aggregation sessions" shows the concept of aggregation membership and Section "Computation of aggregates" outlines how they are used to accurately compute aggregation functions. Section "Realization based on bloom filters" follows with a bloom filter realization of the aggregation memberships. Section "Experimental evaluation" evaluates the performance of DIAS. Section "Discussion and future work" discusses the approach of DIAS and outlines future work. Finally, Section "Conclusions" concludes this paper.

Problem description

Assume an overlay network of nodes, all having an *aggregation value* about the state of a (application) parameter. In this paper, an aggregation value is represented by a numerical (real) value. *Aggregation* is defined in this paper as the computation of aggregation

functions (aggregates), e.g., SUMMATION, by all of the nodes of an overlay network with input the total aggregation values in this overlay network. Aggregation is *decentralized* if it can be performed without using any centralized computational entity for this purpose. Most decentralized aggregation systems have the following features:

- *Function-dependence*: Distributed applications may require the computation of a wide range of aggregation functions. AVERAGE, SUMMATION, MAXIMUM and MINIMUM are common numerical aggregation functions. Textual and rule aggregation are more complex. Aggregation functions share different mathematical properties (Calvo et al. 2002) and, therefore, their computational requirements may vary significantly. Due to this reason, different aggregation methodologies have been developed for specific aggregation functions or classes of aggregation functions. For example, gossip based aggregation (Jelasity et al. 2005) calculates the AVERAGE function as an iterative variance reduction algorithm over the values of nodes in an overlay network. Nonetheless, the COUNT operator that estimates the number of participating nodes cannot be calculated without additional protocol complexity to effectively apply the 'inverse birthday paradox' (Jelasity et al. 2005). The SUMMATION operator is derived by the product estimation of AVERAGE and COUNT and therefore, two instances of gossiping protocols are required. Similar issues are raised (Kempe et al. 2003) together with inaccuracy issues when there are failures in the network.
- *Interaction-dependence*: Most aggregation methodologies are designed in line with the properties, strengths and constraints of the network interaction mechanism that supports them, i.e., gossiping or routing over tree topologies. Replacing the interaction mechanism of an aggregation methodology with a different one makes this methodology inaccurate, cost-ineffective and actually infeasible. The interaction-independence of aggregation methodologies that this paper focuses on concerns the actual option to use a single aggregation mechanism over different interaction mechanisms. However, this abstraction cannot satisfy that the performance of aggregation is comparable between different interaction mechanisms. The variance reduction algorithm applied in gossip-based aggregation (Jelasity et al. 2005) requires gossiping communication between peers in a network. Information diffusion based on which distributed aggregation is performed also depends on a similar gossiping protocol (Kempe et al. 2003). Aggregation over structured topologies, such as trees, relies on multicasting. For example, tree aggregation requires unique paths between nodes in an overlay network to avoid double-counting. This requirement is not satisfied in unstructured (random) overlay networks maintained by gossiping protocols.
- *Static aggregation values*: Aggregation values may change and be derived from a continuous or discrete domain of values. Speed of change matters. Distributed aggregation schemes may be infeasible if aggregation values are highly dynamic. Investigating the degree of tolerable changes in the aggregation values of nodes is crucial for realizing a dynamic aggregation system. Adapting the aggregates with the new aggregation values is potentially a better solution than performing an expensive re-computation.
- *Inaccuracies*: Inaccuracies are estimations of aggregates with significant deviations from the actual aggregates. Two types of inaccuracies are studied:

(i) double-counting and (ii) outdated aggregation values, i.e., values that have changed during runtime. In duplicate-sensitive aggregation functions, such as SUMMATION, summing aggregation values twice results in an inaccurate aggregate. The same holds if aggregation values of nodes in an overlay network change dynamically during system runtime. Aggregates require adaptation to converge to their most recent actual values. Other inaccuracies related to network uncertainties, fault-intolerance etc. are not the focus of this paper and are usually related to the adopted interaction mechanism (Kennedy et al. 2009).

The above features appear to a certain degree in most of the existing aggregation approaches (Ahmed et al. 2006; Haridasan and van Renesse 2008; Jelasity et al. 2005; Kempe et al. 2003; Kashyap et al. 2006; Nath et al. 2008) and are mentioned in the related surveys (Chitnis et al. 2008; Kennedy et al. 2009). These features are actually the limitations of these systems in the sense that they are not generic and adaptive enough to perform aggregation under different network conditions and application requirements. Section "Comparison with related work" discusses and compares these approaches and their limitations in detail. Appendix A summarizes the related aggregation mechanisms discussed in this paper. Motivated by these issues, this paper focuses on the problem of designing a service for dynamic, accurate and decentralized aggregation decoupled from a specific interaction mechanism and aggregation function.

System overview

This paper introduces DIAS, the Dynamic Intelligent Aggregation Service. DIAS is a middleware service that computes aggregation functions from an input set of continuously changing aggregation values distributed in every node of a network. Figure 1 illustrates the three-level architecture of DIAS.

Each level is built by an *aggregator* and *disseminator* agent. These two agents, within a node, provide aggregation values to the agents of other nodes and consume aggregation

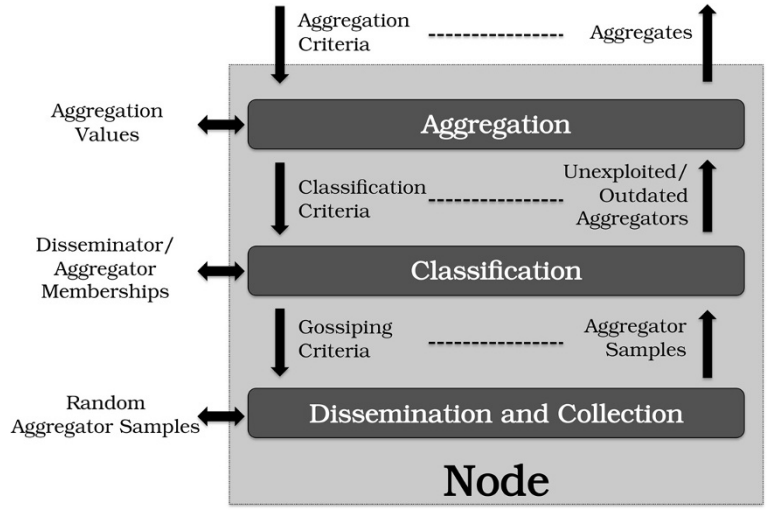

Figure 1 The DIAS middleware architecture.

values from them. However, note that, in practice, applications may not require all agents to disseminate and aggregate values. Section "Discussion and future work" discusses this issue in more detail.

The bottom level of DIAS is responsible for a gossip-based (Jelasity et al. 2007) dissemination and collection of *aggregator* samples. *Disseminators* gossip location information of agents to which the aggregation values need to be sent. Gossiping can be continuously parameterized by gossiping criteria provided by the middle level.

The discovered *aggregator* samples are provided to the middle level in which they are classified. Each *disseminator* classifies the received *aggregators* into three possible classes: (i) exploited, (ii) unexploited and (iii) outdated. These classes indicate if the aggregation value of a *disseminator* has been aggregated before by the classified *aggregators*, if it has not been aggregated or if an earlier (outdated) aggregation value has been aggregated that has changed. Classification is performed based on historical aggregation information generated during runtime[a]. The middle level provides to the top one contact information of possible *aggregators* to which aggregation values can be aggregated. DIAS is able to tune discovery of new aggregation values instead of updating the existing aggregated values and the other way around. These are the *adaptation strategies* of DIAS and are configured by the classification criteria provided by the top level.

Finally, the top level interacts with the remote *aggregators* to exchange aggregation values. These overlay interactions have two possible semantics: exploitation of a new aggregation value or update of aggregates with the most recent aggregation value. A number of aggregates are computed and delivered to the applications as defined by the aggregation criteria.

DIAS addresses the limitations illustrated in Section "Problem description" at a cost of higher communication overhead compared to related methodologies that specialize in specific aggregations functions or interaction mechanisms (Jelasity et al. 2005; Nath et al. 2008). As most of these limitations are related to a lack of abstraction, modularity and customization of aggregation mechanisms, DIAS is designed to split the complexity of dynamic decentralized aggregation into three organizational levels.

Memberships of DIAS are the means to detect inaccuracies such as double-counting and outdated aggregation values. However, a decentralized system cannot explicitly store memberships of all aggregation values locally in each node. This approach is neither scalable, efficient nor decentralized. To overcome this challenge, the probabilistic data structures of *bloom filters* (Bloom 1970) are used in DIAS for management of memberships. Bloom filters provide tremendous space savings at a cost of false positive memberships. DIAS, however, is able to detect false positive inconsistencies and, therefore, maintain a high accuracy level in the computed aggregates without introducing additional communication cost.

Modeling of dynamics

This section introduces a model for aggregation of states. A *state* represents a (aggregation) value of an application parameter at a specific point in time. The state of an application parameter changes during runtime. Decentralized aggregation computes aggregation functions that receive as input the states of different nodes for the same application parameter.

Assume that each of the n nodes of DIAS contains an *aggregator* A_i and a *disseminator* D_i with a *selected state* s'_i that is the one to be aggregated by all nodes. During each runtime iteration, selected state s'_i can be equal to one and only one state from a finite number v of locally unique *possible states* $s'_i = s_i^0|s_i^1|\ldots|s_i^{v-1}$. For example, in a movie recommender system, movies are ranked with one to five stars. The number of stars are the possible states and an actual ranking of a movie is the selected state. Although the possible states in each node are unique, two possible states between different nodes may have the same value. As the selected state changes, an earlier selected state is indicated as $\hat{s}_i$.

The system goal is the aggregation $f(s'_0, s'_1, \ldots, s'_{n-1})$ of all of the selected states in the overlay network during an aggregation phase. An *aggregation phase* is defined as the time period in which the selected states may change but the set of possible states remains the same. During an aggregation phase, the aggregates change continuously as a result of changes in the local selected states. Aggregation does not converge to a single value but rather to a distribution of aggregates over time. Section "Discussion and future work" discusses the applicability of this model in distributed applications.

Dissemination and collection

Decentralized aggregation requires the means to access all of the locations of *aggregators* that acquire the selected states of *disseminators*. Dissemination and collection of *aggregator* samples via gossiping provide lookup in a distributed environment. An *aggregator* sample contains the network identifier of this *aggregator*, e.g., IP address and port number. Each agent of the bottom level maintains its random view that is a list of size r with random *aggregator* samples that are continuously updated via the gossiping protocol of the peer sampling service (Jelasity et al. 2007).

Gossiping provides a highly connected and dynamic overlay network for aggregation. Furthermore, continuous update of the random view enables the discovery of changing aggregation values. The bottom level can be realized with different mechanisms beyond gossiping, e.g., flooding (Jiang et al. 2003), random walks (Gkantsidis et al. 2006) and DHTs (Yuh-Jzer et al. 2005). However, these mechanisms require high customization and DHTs require a topological maintenance. Their utilization becomes more complex within a generic decentralized aggregation service.

Consistent aggregation sessions

The middle level of DIAS provides *aggregators* to the top level that guarantee consistent aggregation sessions. An (unidirectional) *aggregation session* concerns (re)computation of the aggregates by an *aggregator* A_j after the receipt of a selected state from a remote *disseminator* D_i. If (re)computation occurs in both *aggregators* of nodes i and j, this aggregation session is *bidirectional*. An aggregation session is *consistent* if the input selected state of performed (re)computation by an *aggregator* A_j is not (i) a duplicate or (ii) an outdated selected state that has now changed. A consistent aggregation session between an *aggregator* A_j and a *disseminator* D_i is mutually satisfied if and only if the following conditions hold:

- The *disseminator* D_i disseminates for first time (i) its selected state, or (ii) its updated selected state to the *aggregator* A_j.

- The *aggregator* A_j aggregates for first time (i) the selected state, or (ii) the updated selected state of the *disseminator* D_i.

An inconsistent aggregation session usually results in inaccurate aggregates. Note that double-counting does not always result in inaccuracies as some aggregation functions are insensitive to duplicates, i.e., MAXIMUM or MINIMUM. However, duplicates cause additional communication and processing overhead in nodes. For this reason, this paper treats inconsistent aggregation sessions as subject of prevention.

Selecting *aggregators* that result in consistent aggregation sessions requires some form of history information about the past aggregation sessions performed. This section introduces the concept of aggregation memberships and their use to classify *aggregators* in the outdated, exploited and unexploited classes. Beyond consistency, this classification provides the option to perform the update of aggregates in favor of (i) changing (outdated) aggregation values or (ii) unexploited aggregation values. These two options distinguish the two adaptation strategies of DIAS.

Note that classification is used as the means to guarantee consistent aggregation sessions that enable a more generic design for aggregation in order to overcome the limitations illustrated in Section "Problem description".

Aggregation memberships

If an arbitrary aggregation value is selected from the network during an aggregation phase, this aggregation value has a probability of membership in the computed aggregates. Aggregation membership $M_{group}(member)$ of a certain 'member' to a certain 'group' is either positive or negative. This concept can be applied to the aggregation dynamics illustrated in Section "Modeling of dynamics". Each agent of the middle level in a node i stores unique identifiers of possible states $\mathsf{S}_i^0, \ldots, \mathsf{S}_i^{\nu-1}$ corresponding to the actual possible states $s_i^0, \ldots, s_i^{\nu-1}$. Respectively, S'_i and $\hat{\mathsf{S}}_i$ refer to the unique identifiers of the selected s'_i and outdated $\hat{s}_i$ state in node i. The middle level stores a representation of the local states, their unique identifiers, and the top level stores the actual states, e.g., numerical or other type. The middle level also uses the local unique network identifier of the node to map the local *aggregator* A_i and *disseminator* D_i. Therefore, $\mathsf{A}_i = \mathsf{D}_i$. The following four aggregation memberships are defined in a unidirectional aggregation session between an *aggregator* A_j and a *disseminator* D_i in two nodes i and j:

Membership 1 ($M_{\mathsf{D}_i}(\mathsf{A}_j)$)**.** *An aggregator in a disseminator.*

A *disseminator* D_i stores the identifier of an *aggregator* A_j to which it has disseminated its selected state at least once during an aggregation phase.

Membership 2 ($M_{\mathsf{S}_i^u}(\mathsf{A}_j)$)**.** *An aggregator in a possible state.*

A *disseminator* D_i stores the identifier of an *aggregator* A_j for each possible state identified as S_i^u aggregated by this *aggregator*.

Membership 3 ($M_{\mathsf{A}_j}(\mathsf{D}_i)$)**.** *A disseminator in an aggregator.*

An *aggregator* A_j stores the identifier of a *disseminator* D_i from which it has aggregated its selected state at least once during an aggregation phase.

Membership 4 ($M_{\mathsf{A}_j}(\mathsf{S}'_i)$)**.** *A selected state in an aggregate.*

An *aggregator* A_j stores the identifier of a selected state S'_i aggregated from a *disseminator* D_i.

Figure 2 illustrates the aggregation memberships of DIAS stored in the middle level. Aggregation memberships can be used as follows: Assume an aggregation session between *disseminator* D_i that sends its selected state S'_i and *aggregator* A_j that receives this state. *Disseminator* D_i knows if *aggregator* A_j aggregates its selected state S'_i for first time by checking aggregation membership $M_{\mathsf{D}_i}(\mathsf{A}_j)$. Furthermore, D_i knows if A_j has aggregated a different possible state earlier by checking all aggregation memberships $M_{S_i^u}(\mathsf{A}_j)$. Respectively, *aggregator* A_j knows if it has aggregated a selected state from *disseminator* D_i by checking aggregation membership $M_{\mathsf{A}_j}(\mathsf{D}_i)$. Moreover, A_j knows if the specific selected state S'_i has been aggregated earlier by checking aggregation membership $M_{\mathsf{A}_j}(\mathsf{S}'_i)$. Therefore, both (i) duplicate and (ii) outdated selected states can be detected between an *aggregator* and a *disseminator* and the consistency of an aggregation session is satisfied.

Aggregation memberships represent two mutual conditions resulting in information redundancy: Both *aggregators* and *disseminators* store membership information about their in-between aggregation. Section "Realization based on bloom filters" shows how this redundancy is exploited in an efficient model realization of aggregation memberships based on bloom filters.

Classification

Classification performed in the middle level is based on an *aggregation pool* containing three *aggregation views*. These views are queues of a limited size in which *aggregators* are classified. Three aggregation views are defined in the aggregation pool: (i) exploited, (ii) unexploited and (iii) outdated. The exploited *aggregators* of a *disseminator* D_i are the ones that have aggregated its earliest selected state s'_i. The unexploited *aggregators* of a *disseminator* D_i are the ones with which a consistent aggregation session has not been established. Finally, the outdated *aggregators* of a *disseminator* D_i are the ones that have

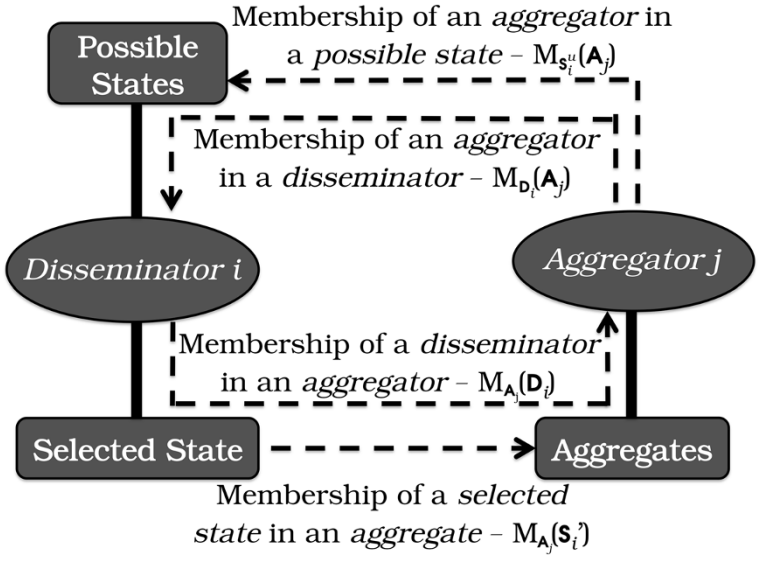

Figure 2 The aggregation memberships of DIAS.

aggregated a selected state of this *disseminator* earlier but since then this selected state has changed. Aggregation views are used as a buffer and have a limited size to allow scalability and decentralization.

Algorithm 1 illustrates the classification of an *aggregator* A_j in the aggregation pool based on the aggregation memberships $M_{\mathsf{D}_i}(\mathsf{A}_j)$ and $M_{\mathsf{S}'_i}(\mathsf{A}_j)$ of a *disseminator* D_i. When A_j is received by the bottom level, the middle level executes a membership query $M_{\mathsf{D}_i}(\mathsf{A}_j)$ that indicates if a consistent aggregation session has been performed between A_j and D_i. If membership is negative, *aggregator* A_j is classified as unexploited. Otherwise, if membership is positive, the next membership query $M_{\mathsf{S}'_i}(\mathsf{A}_j)$ is performed to indicate if *aggregator* A_j has computed in its aggregates the most recent selected state S'_i. If this membership is positive, *aggregator* A_j is exploited (duplicate aggregation value), otherwise, *aggregator* A_j has computed an earlier selected state of D_i and therefore A_j is classified as outdated.

Algorithm 1 Classification in the aggregation pool.

Require: *aggregator* A_j from bottom level
1: **if** $M_{\mathsf{D}_i}(\mathsf{A}_j)$: *negative* **then**
2: A_j is unexploited
3: **else if** $M_{\mathsf{S}'_i}(\mathsf{A}_j)$: *negative* **then**
4: A_j is outdated
5: **else**
6: A_j is exploited
7: **end if**

If the selected state of *disseminator* D_i changes, the aggregation pool requires rearrangement. *Aggregators* contained in the exploited view before the change of the selected state move to the outdated view. In contrast, *aggregators* contained in the outdated view before the change of the selected state are queried again ($M_{\mathsf{S}'_i}(\mathsf{A}_j)$) and are classified as outdated or exploited. As a result of this querying, the aggregation pool remains consistent and adapts instantly after a change of the selected state.

Adaptation strategies

A consistent aggregation session is established with either an unexploited or an outdated *aggregator*. Priority is defined by the classification criteria received from the top level. These two options are the two adaptation strategies of DIAS and are referred to as EXPLOITATION and UPDATE respectively.

EXPLOITATION is a more relevant adaptation strategy if selected states do no change often and the aggregates still converge to their actual values, for example, at the beginning of aggregation or during network scaling with new nodes. In contrast, UPDATE is more relevant for steady size of networks and when aggregates have converged to the actual values. Changes of the selected states after convergence require adaptations of aggregates.

Selection of *aggregators* from the aggregation pool is conditional to the availability of *aggregators* in the class of preference for each adaptation strategy. This means that if EXPLOITATION is adopted but the unexploited view of the aggregation pool is empty, then outdated *aggregators* are selected corresponding to the selections of the UPDATE strategy.

The same holds if the UPDATE strategy is adopted and the view of outdated *aggregators* is empty: unexploited *aggregators* are selected. To this extent, the adaptation strategies of DIAS are dynamic.

Adoption of an adaptation strategy can be static, e.g., a system parameter contained in the classification criteria, or dynamic during system runtime. For example, the adopted adaptation strategy may change based on monitored parameters or based on a time period that aggregates do not change significantly.

Aggregation session

An aggregation session requires remote interactions between *aggregators* and *disseminators* to guarantee its consistency. Figure 3 illustrates these interactions between a node i and j during a bidirectional aggregation session. A unidirectional aggregation session is established with two messages depicted by the arrows (1) and (2). A bidirectional aggregation session includes a third message depicted by arrow (3).

The 'request' message, illustrated by arrow (1), initiates an aggregation session and contains the following information:

- Flag: This denotes a unidirectional 'uni' or bidirectional 'bi' aggregation session.
- Class: This denotes if the *aggregator* A_i, receiving this message, is classified by a *disseminator* D_j as unexploited or outdated.
- D_j: This is the identifier of the *disseminator* D_j that has performed the classification of the *aggregator* A_i.
- S'_j: This is the selected state identifier of D_j.
- S_j: This is the earlier selected state identifier of the *disseminator* D_j aggregated by A_i.

A 'response' message, illustrated by arrow (2) or (3), completes a unidirectional or bidirectional aggregation session and contains the following information:

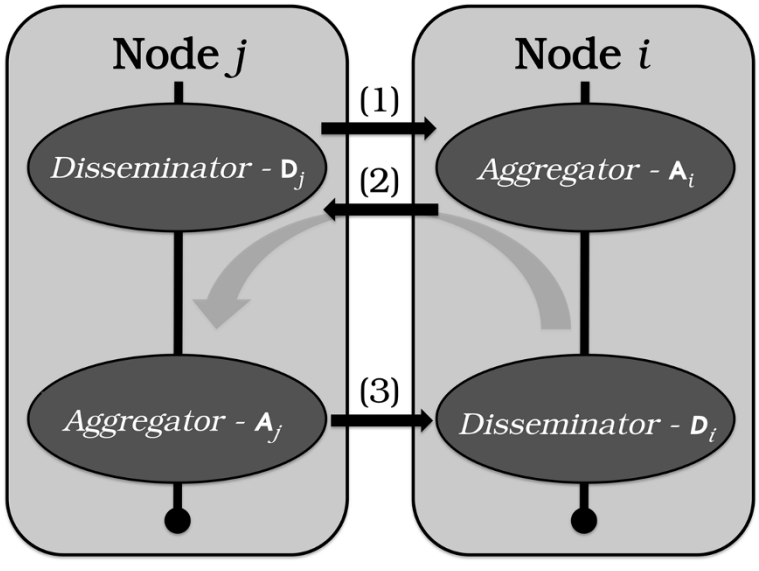

Figure 3 The remote interactions of DIAS.

- Flag: This denotes a unidirectional 'uni' or bidirectional 'bi' aggregation session. A third flag, the 'uni-bi', denotes the upgrade of a unidirectional aggregation session to a bidirectional one by including a 'request' message flagged as 'bi'.
- Class: This denotes if the *aggregator* A_j, sending this message, is classified by a *disseminator* D_i as unexploited or outdated.
- A_j: This is the identifier of the *aggregator* A_j.
- 'Request' message: This integrated message is optional. It upgrades the unidirectional aggregation session to a bidirectional one.

Note that the integrated 'request' message in the 'response' message provides one message fewer for a bidirectional session to complete.

Computation of aggregates

The top level is responsible for the computation of aggregates. An aggregate is continuously computed based on an aggregation function provided by the aggregation criteria. Aggregates are updated by sending the value of the selected state to *aggregators* provided by the middle level and classified as unexploited. If the provided *aggregators* are classified as outdated, the earlier selected state is sent as well.

The top level forms an overlay network between *aggregators* and *disseminators* linked with overlay links that have two possible semantic values: unexploited or outdated but not exploited. Therefore, the computed aggregation functions exclude overlay links from the top level that result in duplicates (exploited *aggregators*). The aggregation memberships, the classification, the selection of *aggregators* are all complexity hidden from the aggregation process of the top level. As explained in Section "Adaptation strategies", the adaptation strategies tune the aggregation process in favor of (i) updating aggregates with the most recent selected states (UPDATE) or (ii) discovering new selected states (EXPLOITATION). The top level has to only provide the classification criteria that trigger this optimization and inform about changes in the selected state.

Delivery of aggregates to applications may be performed periodically. Another option is a minimum deviation threshold over a certain time period that denotes convergence to the actual aggregate values. The aggregation criteria define these requirements.

Realization based on bloom filters

Explicit storage of aggregation memberships in every agent of the middle level is not a scalable, efficient and decentralized solution. Aggregation memberships can be a cost-effective and viable approach in large-scale decentralized environments by using an implicit storage mechanism: bloom filters (Bloom 1970).

A bloom filter is a probabilistic data structure for efficient membership storage and querying. A bloom filter is based on a number of k hash functions that hash an element in a limited binary space of 2^{m} size, where m is the size of the bit vector in which information is stored. More specifically, each hash function outputs a random index in this binary space.

A *simple bloom filter* supports insertions and membership queries. During an insertion, the bits that are indexed by the hash functions are set to 1. During membership queries, the membership of an element in the bloom filter is confirmed if all of the bits indexed by all of the hash functions are 1.

Counting bloom filters additionally support removal of memberships (Li et al. 2000). This is achieved by representing the storage space with integers, instead of single bits, that act as counters. Insertions increment the counters indexed by hash functions and removals decrement respectively. Data overflow by consecutive insertions is prevented by choosing an adequate size of 3 - 4 bits for the integers. Therefore, a counting bloom filter is 3 - 4 times larger than a simple one.

Each of the memberships illustrated in Figure 2 is stored in a bloom filter. More specifically, a *disseminator* D_i has a simple bloom filter for storing $M_{\mathsf{D}_i}(\mathsf{A}_j)$ memberships and v counting bloom filters, one for each possible state, for storing $M_{\mathsf{S}_i^u}(\mathsf{A}_j)$ memberships. The counting bloom filters provide the flexibility to reflect the changes of the selected states. For example, in an aggregation session between a *disseminator* D_i and an outdated *aggregator* A_j, the membership $M_{\hat{\mathsf{S}}_i'}(\mathsf{A}_j)$ is removed from the counting bloom filter of the earlier selected state $\hat{\mathsf{S}}_i$ and the membership $M_{\mathsf{S}_i'}(\mathsf{A}_j)$ is added in the counting bloom filter of the most recent selected state S_i'. Complementarily, the *aggregator* A_j has a simple bloom filter for storing the $M_{\mathsf{A}_j}(\mathsf{D}_i)$ memberships and a counting bloom filter for storing the $M_{\mathsf{A}_j}(\mathsf{S}_i')$ memberships. This provides a consistent update of aggregates by replacing textsfoutdated selected states with the most recent ones.

The space saving achieved by bloom filters come at the cost of false positives. False positive membership indicates that a state or agent identifier is hashed in a bloom filter when it is actually not hashed. The probability of false positives depends on (i) the number of elements stored in the bloom filter, (ii) the number k of hash functions and (iii) the size 2^{m} of the storage space. The minimum number of bits in a simple bloom filter x that hashes n elements and results in a certain probability $P_{fp}(x)$ of false positives is computed as $2^{\mathsf{m}} = -n\frac{\ln P_{fp}(x)}{(\ln 2)^2}$ (Deke et al. 2010). False positives can cause inconsistent aggregation sessions (inaccurate aggregates) and additional communication overhead if they are not detected and eliminated.

The space savings computed for a bloom filter can be outlined as follows: Assume at least $128n$ bits stored in conventional data structures such as an array. The $128n$ bits are actually n number of agent or state memberships represented by global unique identifiers of 128 bits. A hash table requires even a higher storage space due to the additional storage of indexes that enhance searching operations. In contrast, assume a bloom filter x with a probability $P_{fp}(x) = 0.01$ of false positives that stores the same number n of memberships. The relation $2^{\mathsf{m}} = -n\frac{\ln P_{fp}(x)}{(\ln 2)^2}$ shows that, in this case, an array stores $128/9.6 \approx 13$ times the space of this bloom filter. For a bloom filter with $P_{fp}(x) = 0.1$ and $P_{fp}(x) = 0.001$, its space storage is approximately 56 and 9 times lower respectively.

Note that false negatives in counting bloom filters may occur if an erroneous element removal is performed. This removal may result in a biased and inconsistent probabilistic data structure. For example, if a removed element is not actually hashed, then its removal changes bits indicating memberships of other elements that are actually hashed (Deke et al. 2010). This paper assumes that false negatives cannot be generated in principle if and only if removals are not performed from counting bloom filters. Otherwise, Section "Second level check" illustrates how false negatives are prevented in DIAS if removals are performed.

The mutual membership check

DIAS deals with the problem of false positives in bloom filters by taking advantage of decentralized mutual membership checks between *disseminators* and *aggregators*. A *mutual memberships check*, denoted as '⋒' in this paper, is the process of querying two memberships in a *disseminator* and an *aggregator* that are assumed to either be both present or not. For example, the aggregation memberships $M_{\mathsf{D}_i}(\mathsf{A}_j)$ and $M_{\mathsf{A}_j}(\mathsf{D}_i)$ are mutual. During an aggregation phase, a *disseminator* stores memberships of *aggregator* identifiers and, respectively, these *aggregators* store memberships of the respective *disseminator* identifiers resulting in mutual aggregation memberships. $M_{\mathsf{S}'_i}(\mathsf{A}_j)$ and $M_{\mathsf{A}_j}(\mathsf{S}'_i)$ are also mutual memberships. Selected state S'_i of a *disseminator* D_i is associated with the $M_{\mathsf{S}'_i}(\mathsf{A}_j)$ membership of an *aggregator* A_j. Respectively, *aggregator* A_j stores the $M_{\mathsf{A}_j}(\mathsf{S}'_i)$ membership of the selected state identifier S'_i.

Mutual membership checks provide detection of false positives in the bloom filters of DIAS. Only if multiple false positives occur between $M_{\mathsf{D}_i}(\mathsf{A}_j)$-$M_{\mathsf{A}_j}(\mathsf{D}_i)$ and $M_{\mathsf{S}'_i}(\mathsf{A}_j)$-$M_{\mathsf{A}_j}(\mathsf{S}'_i)$ in a single aggregation session, then an inconsistent aggregation session may come as a result of these false positives.

Assume two arbitrary memberships $M_x(a)$ and $M_y(b)$ based on the unique identifiers of two members a and b in the groups x and y respectively. Assume also that these two memberships are mutual, meaning that they should be both positive or negative such as $M_x(a) \Cap M_y(b) : \textit{positive} \mid M_x(a) \Cap M_y(b) : \textit{negative}$. $M_x(a)$ and $M_y(b)$ are stored in two simple bloom filters with false positive probabilities $P_{fp}(x)$ and $P_{fp}(y)$ respectively. The possible outcomes of the mutual membership check are the following:

> **Check 1.** ***if*** $M_x(a)$ *: positive* ***and*** $M_y(b)$ *: positive* ***then*** $M_x(a) \Cap M_y(b)$ *: positive*

$M_x(a)$ and $M_y(b)$ memberships are confirmed with a probability of $1 - P_{fp}(x)P_{fp}(y)$. This confirmation is false if and only if both bloom filters generate a false positive that is the product $P_{fp}(x)P_{fp}(y)$ of their false positive probabilities.

> **Check 2.** ***if*** $M_x(a)$ *: positive* ***and*** $M_y(b)$ *: negative,* ***or****,* $M_x(a)$ *: negative* ***and*** $M_y(b)$ *: positive* ***then*** $M_x(a) \Cap M_y(b)$ *: negative*

$M_x(a)$ and $M_y(b)$ memberships are not confirmed with a probability of 1. In this case, one of the bloom filters generates a false positive.

> **Check 3.** ***if*** $M_x(a)$ *: negative* ***and*** $M_y(b)$ *: negative* ***then*** $M_x(a) \Cap M_y(b)$ *: negative*

$M_x(a)$ and $M_y(b)$ memberships are not confirmed with a probability of 1.

Mutual membership checks provide (i) a decrease in the probability that an inconsistent aggregation session occurs (Check 1) and (ii) detection of false positives (Check 2). This section introduces a consistency mechanism of aggregation sessions for accurate aggregates. This mechanism is based on two nested mutual membership checks between the bloom filters of an *aggregator* A_j and a *disseminator* D_i that define the four possible *outcomes* of an aggregation session:

- Exploitation: *Aggregator* A_j and *disseminator* D_i are involved for a first time in a consistent aggregation session as defined in Section "Consistent aggregation sessions". A selected state has not been aggregated before and the aggregates are

updated with new information. The $M_{\mathsf{D}_i}(\mathsf{A}_j)$, $M_{\mathsf{A}_j}(\mathsf{D}_i)$, $M_{\mathsf{S}'_i}(\mathsf{A}_j)$ and $M_{\mathsf{A}_j}(\mathsf{S}'_i)$ memberships are added in the respective bloom filters.

- Update: *Aggregator* A_j and *disseminator* D_i have been involved before in a consistent aggregation session, however, this time the selected state has changed. The *aggregator* A_j updates its aggregates with the new selected state. The $M_{\hat{\mathsf{S}}_i}(\mathsf{A}_j)$ membership is replaced by the $M_{\mathsf{S}'_i}(\mathsf{A}_j)$ membership and $M_{\mathsf{A}_j}(\hat{\mathsf{S}}_i)$ is replaced by $M_{\mathsf{A}_j}(\mathsf{S}'_i)$.
- Duplicate: *Aggregator* A_j and *disseminator* D_i have been involved before in an aggregation session with the same selected state. Aggregation is not performed.
- Inconsistency: *Aggregator* A_j and the *disseminator* D_i are involved for a first time in a consistent aggregation session but the mutual membership check cannot confirm this. Alternatively, *aggregator* A_j and *disseminator* D_i have been involved before in an aggregation session with a different selected state. However, the consistency check cannot identify the textsfoutdated selected state to replace. These uncertainties are treated as an inconsistency and are a result of multiple false positives in the bloom filters.

The two nested mutual membership checks illustrated in Section "First level check" and "Second level check" show how an aggregation session reaches each of the above possible outcomes. The results of the memberships are exchanged in the messages defined in Section "Aggregation session".

First level check

This mutual membership check identifies if a consistent aggregation session has not been performed between an *aggregator* A_j and a *disseminator* D_i. *Disseminator* D_i queries the $M_{\mathsf{D}_i}(\mathsf{A}_j)$ membership of the A_j identifier in its bloom filter. Complementarily, *aggregator* A_j queries $M_{\mathsf{A}_j}(\mathsf{D}_i)$ membership. The $M_{\mathsf{D}_i}(\mathsf{A}_j)$ and $M_{\mathsf{A}_j}(\mathsf{D}_i)$ memberships are mutual as they are either both added in the bloom filters or not. Therefore, a mutual membership check provides the following benefits at the first level of the nested mutual membership check: (i) A decrease in the probability of an inconsistent aggregation session that requires two false positives generated by the two bloom filters. (ii) Detection of a false positive in either the $M_{\mathsf{D}_i}(\mathsf{A}_j)$ or $M_{\mathsf{A}_j}(\mathsf{D}_i)$ membership. Algorithm 2 illustrates the first level of the nested mutual membership check.

Algorithm 2 The first level check.

Require: access to *disseminator* D_i and *aggregator* A_j

1: **if** $M_{\mathsf{D}_i}(\mathsf{A}_j) \Cap M_{\mathsf{A}_j}(\mathsf{D}_i) : \mathit{negative}$ **then**
2: outcome=exploitation
3: **else**
4: go to Algorithm 3
5: **end if**

Ensure: outcome

This mutual membership check detects an exploitation outcome in an aggregation session if and only if $M_{\mathsf{D}_i}(\mathsf{A}_j) \Cap M_{\mathsf{A}_j}(\mathsf{D}_i) : \mathit{negative}$. This outcome is generated if at least one of the $M_{\mathsf{D}_i}(\mathsf{A}_j)$ and $M_{\mathsf{A}_j}(\mathsf{D}_i)$ memberships, in case of a single false positive, or both memberships, in case of no false positives, cannot be confirmed. On this first level, the

exploitation outcome is reached with an absolute certainty. However, two simultaneous false positives in the $M_{\mathsf{D}_i}(\mathsf{A}_j)$ and $M_{\mathsf{A}_j}(\mathsf{D}_i)$ memberships are possible. Therefore, further examination is required on a second level of a mutual membership check to detect multiple false positives and lower the uncertainties of the outcomes.

Second level check

The second level of the mutual membership check detects if there is an textsfoutdated selected state $\hat{S}_i$ aggregated from a *disseminator* D_i that differs from its new selected state S'_i. The detection is performed by querying every $M_{\mathsf{S}_i^u}(\mathsf{A}_j)$ bloom filter membership of the respective possible state $\mathsf{S}_i^u \in \left\{\mathsf{S}_i^0, \ldots, \mathsf{S}_i^{v-1}\right\}$. $M_{\mathsf{A}_j}(\mathsf{S}_i^u)$ membership is also queried for every possible state S_i^u. The number o of positive mutual memberships $M_{\mathsf{S}_i^u}(\mathsf{A}_j) \Cap M_{\mathsf{A}_j}(\mathsf{S}_i^u)$ define the outcome of an aggregation session as illustrated in Algorithm 3.

Algorithm 3 The second level check.

Require: access to *disseminator* D_i and *aggregator* A_j

1: $o = 0, \hat{\mathsf{S}}_i = \mathsf{S}'_i$
2: **for** $u = 0$ to $v - 1$ **do**
3: **if** $M_{\mathsf{S}_i^u}(\mathsf{A}_j) \Cap M_{\mathsf{A}_j}(\mathsf{S}_i^u) : positive$ **then**
4: $\hat{\mathsf{S}}_i = \mathsf{S}_i^u$
5: $o = o + 1$;
6: **end if**
7: **end for**
8: **if** $o = 0$ **then**
9: outcome=exploitation
10: **else if** $o = 1$ **then**
11: **if** $\hat{\mathsf{S}}_i \neq \mathsf{S}'_i$ **then**
12: outcome=update
13: **else**
14: outcome=duplicate
15: **end if**
16: **else** $\{o > 1\}$
17: outcome=inconsistency
18: **end if**

Ensure: outcome

If there are no positive mutual memberships detected ($o = 0$ in line 8 and 9 of Algorithm 3), there is no positive $M_{\mathsf{A}_j}(\mathsf{S}_i^u)$ membership (no selected state aggregated before from D_i) and/or there is no positive $M_{\mathsf{S}_i^u}(\mathsf{A}_j)$ membership in any bloom filter of the possible states. This condition conflicts with the positive result of the mutual membership check $M_{\mathsf{D}_i}(\mathsf{A}_j) \Cap M_{\mathsf{A}_j}(\mathsf{D}_i)$ in the first level. Both $M_{\mathsf{D}_i}(\mathsf{A}_j)$ and $M_{\mathsf{A}_j}(\mathsf{D}_i)$ memberships are false positives. The outcome in this case is an exploitation.

If there is one positive mutual membership detected ($o = 1$ in lines 10-15), the system can derive the textsfoutdated selected state $\hat{\mathsf{S}}_i$. The outcome is either a duplicate, if the textsfoutdated selected state $\hat{\mathsf{S}}_i$ is the same with the new selected state S'_i, or an update

in the opposite case. The uncertainty of this outcome is minimized by the nested mutual membership checks.

Finally, if more than one positive mutual membership is detected ($o > 1$ in lines 16-18), multiple false positives occur that cannot be identified. These false positives concern $M_{S_i^u}(\mathsf{A}_j)$ and $M_{\mathsf{A}_j}(\mathsf{S}_i^u)$, or $M_{\mathsf{D}_i}(\mathsf{A}_j)$ and $M_{\mathsf{A}_j}(\mathsf{D}_i)$ in the first level of the nested mutual membership check. The outcome is an inconsistency and therefore, any aggregation at this point may result in inaccuracies of the aggregates.

The 'safer' approach to handle inconsistencies is to ignore these aggregation sessions and not perform any aggregation that may result in inaccurate aggregates. However, not only the aggregates can be influenced in this case. Recall from the beginning of this section that removal of a membership from a counting bloom filter that is actually not present introduces false negatives (Deke et al. 2010). Therefore, the following aggregation sessions are prone to inaccuracies as the assumption of no false negatives does not hold anymore. By skipping inconsistent aggregation sessions, DIAS makes sure that the condition of no false negatives in counting bloom filters is not violated.

Experimental evaluation

DIAS is implemented and evaluated in Protopeer (Galuba et al. 2009), a prototyping toolkit for distributed systems. The experimental settings illustrated in this section are summarized in Appendix B. A network of $n = 1500$ nodes runs DIAS for $t(\mathsf{DIAS}) = 800$ epochs. The agents of each node act both as *aggregators* and *disseminators*. Each epoch lasts for $T(\mathsf{DIAS}) = 1000$ ms that is the default parameter value in Protopeer. In practice, the selection of this parameter is performed based on factors such as the available bandwidth in the network. The system initially bootstraps a ring topology. The bootstrapping period is $t'(\mathsf{DIAS}) = 6$ epochs and the size of the ring view is $|\mathbf{v}(ring)| = 5$ for each node.

A simulated application of dynamically changing states is bootstrapped in $t'(application) = 15$ epochs. Each application instance in each node generates $\nu = 5$ numerical possible states during each aggregation phase. The possible states are selected randomly from the range $[0, 1)$ defined by five different beta distributions, one for each possible state. Appendix C illustrates these beta distributions. The selected state changes cyclically as $s_i' = s_i^0, s_i^1, \ldots, s_i^{\nu-1}, s_i^0$, etc. Two factors trigger these changes: (i) time and (ii) the parameter itself that the possible states represent. These factors are modeled based on two probabilities: (i) the probability $P_c(time)$ of changing a selected state every period $T(application)$ and (ii) the probability $P_c(parameter)$ of change in a specific type of application parameter. The probability $P_c(s_i')$ of a node i to change its selected state is $P_c(s_i') = P_c(time)P_c(parameter)$ assuming that the two probabilities $P_c(time)$ and $P_c(parameter)$ are independent.

Two types of changes in the selected states are examined: synchronous and asynchronous. In *synchronous changes*, the selected states of all nodes in the network change simultaneously. Synchronous changes are modeled as $P_c(time) = 1$ and $P_c(parameter) = 1$ for $T(application) = 200$ epochs. In contrast, *asynchronous changes* occur arbitrary over time. A dynamic setting of asynchronous changes is modeled as $P_c(time) = 0.4$ and $T(application) = 0.7$ for $T(application) = 10$ epochs. In practice, the changes in the selected states depend on the dynamics of the application.

The execution period of the top level is synchronized with the one of the middle level as $T(top) = T(middle) = 1000$ ms. The AVERAGE, SUMMATION and MAXIMUM aggregation functions are computed. The messages exchanged by the middle and top level are integrated. This minimizes the number of exchanged messages $\lambda(sessions)$ to the three ones illustrated in Section "Aggregation session". The integrated messages additionally contain the actual states for the computation of the aggregation functions. The aggregates are provided to the application after every computation.

The middle level is periodically executed at $T(middle) = 1000$ ms during which $z = 10$ bidirectional aggregation sessions are initiated at maximum. The size of the aggregation pool is selected to $q = 3 * 15 = 45$ with each of the unexploited, exploited and outdated containing 15 *aggregators* at maximum. The aggregation pool is filled by classifying $e = 15$ random *aggregator* samples collected from the bottom level in each execution period. Static adoptions of the EXPLOITATION and UPDATE strategies are evaluated.

Aggregation memberships are realized in the bloom filters of the XSiena BloomFilter library (Jerzak and Fetzer 2008). Double hashing (Dillinger and Manolios 2004) is used for collision resolution in the hashed elements of bloom filters. The size 2^{m} of the bloom filters and the number of hash functions k are selected empirically using the testing tools of XSiena BloomFilter. The expected number of hashed elements during the performed experiments is equal to the network size n. This selection is performed manually during system parameterization or in an automated fashion. In the latter case, DIAS is initialized with a default size of bloom filters and computes the system size using the COUNT aggregation function.

Three schemes are adopted in DIAS: (i) m = 16, k = 24, (ii) m = 14, k = 24 and (iii) m = 14, k = 6. The first scheme, with $2^{16} = 65536$ bits = 8.192 KB, does not result in false positives during the performed library tests, whereas false positives appear in the other two schemes because of the fewer number of bits available for hashing: $2^{14} = 16384$ bits = 2.048 KB. The relation $2^m = -n\frac{\ln p}{(\ln 2)^2}$ verifies the probability of false positives. For $n = 1500$, the probability of false positives in the first scheme is $0.76 * 10^{-9}$, whereas, for the other two schemes is 0.005. The second scheme introduces higher randomness compared to the third one due to the higher number of hash functions. However, the second scheme causes a higher number of bit changes during insertions. This results in a higher number of potential collisions (Dillinger and Manolios 2004) that cause a higher number of false positives.

The bottom level is realized by the peer sampling service (Jelasity et al. 2007). The size of the random view is $r = 50$ and the execution period is $T(bottom) = T(\mathsf{DIAS})/5 = 250$ ms. The values of the 'view selection', 'view propagation' and 'peer selection' policies (Jelasity et al. 2007) are selected to maximize the randomness and dissemination speed.

The efficiency of DIAS is related to how close the values of the computed aggregates are to the actual ones. This closeness is quantified by two evaluation metrics: (i) accuracy α and (ii) matching μ. *Accuracy* α is defined as $\alpha = 1 - \varepsilon/\varepsilon_{max}$ where ε is the *absolute error* and ε_{max} is the *maximum probable absolute error*. The absolute error is the absolute difference of the actual aggregate from the computed aggregate. The maximum probable absolute error is the maximum possible absolute difference that the actual aggregate and the computed aggregate can have. Note that the convergence of accuracy is particularly interesting for the evaluation of DIAS as it outlines its speed and adaptivity in the

computed aggregates. *Matching* μ is based on the calculation of the correlation coefficient and indicates the closeness of the distribution of the computed aggregates to the distribution of the actual aggregates. This metric is especially useful for the evaluation of DIAS under asynchronous changes.

The source data from which accuracy is computed are illustrated in Appendix C. Accuracy and matching are studied in line with the communication cost of the aggregation sessions in terms of the number of messages $\lambda(sessions)$ exchanged. The communication cost of the bottom level is excluded from the illustrated results as it is constant (Jelasity et al. 2007). The results are interpreted based on the number of aggregation outcomes that aggregation sessions result in. Finally, the effect of (i) the size of aggregation pool, (ii) the size of aggregation classes, (iii) the number of *aggregator* samples, (iv) the number of aggregation sessions (v) and the periodical executions are factors that are experimentally evaluated by (Pournaras 2013).

Adaptation strategies

This section evaluates the efficiency of DIAS with and without adaptation strategies. For this reason, the bloom filter scheme of m = 16 and k = 24 is adopted that does not result in false positives. The case when DIAS does not employ adaptation strategies is referred to as the RANDOM strategy and concerns random *aggregator* samples without a classification in the aggregation pool.

Figures 4a-4c illustrate the accuracy convergence under synchronous changes. EXPLOITATION and UPDATE converge to the maximum accuracy $\alpha = 1$ and adapt the aggregates within 100 epochs. Matching μ is 0.79, 0.59 and 0.90 for AVERAGE, SUMMATION and MAXIMUM respectively. The distribution of the exploitation and update outcomes depicted in Figure 5a and 5b explains the convergence of accuracy. These outcomes represent $z = 10$ bidirectional aggregations sessions by $n = 1500$ *aggregators*: $10 * 2 * 1500 = 30000$ exploitation and update outcomes. Note that, under synchronous changes, EXPLOITATION and UPDATE have the same effect. The total number of aggregation sessions with an exploitation outcome are performed within the first $T(application) = 200$ epochs. The next aggregation sessions result in update outcomes.

RANDOM also achieves a high accuracy according to Figures 4a-4c, with 0.71, 0.33 and 0.90 matching μ for each aggregate respectively. However, RANDOM has a slower convergence of 150 additional epochs compared to EXPLOITATION and UPDATE. This is because of the number of duplicate outcomes that reaches 28000 during convergence as depicted in Figure 5c. EXPLOITATION and UPDATE do not cause duplicate outcomes as the exploited *aggregators* are not selected from the aggregation pool.

Figure 4d-4f illustrate the convergence of accuracy under asynchronous changes. Although $P_c(time)P_c(parameter)n = 0.4 * 0.7 * 1500 = 420$ selected states change on average every $T(application) = 10$ epochs, accuracy converges to the maximum. Matching μ between the actual and computed AVERAGE for EXPLOITATION and UPDATE is 0.57 and 0.70 respectively. RANDOM is not influenced significantly with a matching of $\mu = 0.66$ for AVERAGE. RANDOM reaches exploitation and update outcomes during the converge period in contrast to EXPLOITATION that mostly reaches exploitation outcomes in the first 100 epochs (Figure 5d) and update outcomes in the next epochs (Figure 5e). Similarly with the case of synchronous changes, RANDOM requires 150 additional epochs to converge compared to EXPLOITATION. A converged number of 10000 duplicate

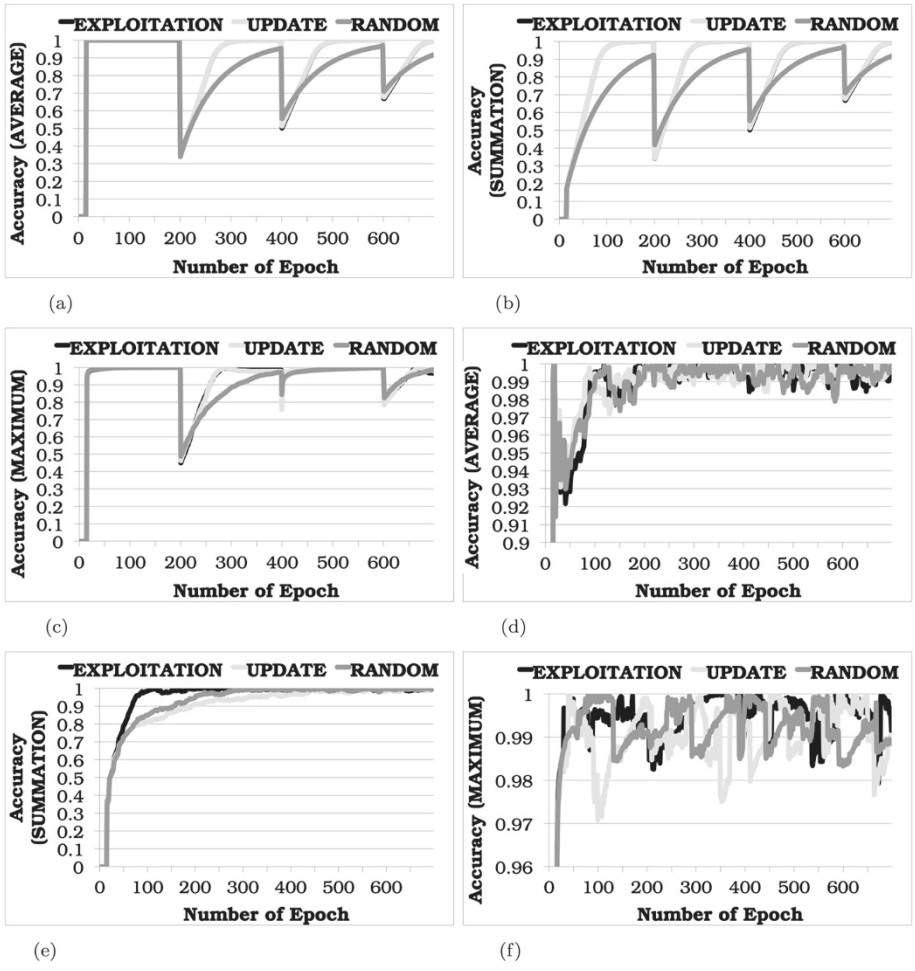

Figure 4 Accuracy of adaptation strategies. (a) AVERAGE under synchronous changes. **(b)** SUMMATION under synchronous changes. **(c)** MAXIMUM under synchronous changes. **(d)** AVERAGE under asynchronous changes. **(e)** SUMMATION under asynchronous changes. **(f)** MAXIMUM under asynchronous changes.

outcomes depicted in Figure 5f causes this delay. Matching μ in MAXIMUM is 0.67, 0.55 and 0.45 respectively for EXPLOITATION, UPDATE and RANDOM. SUMMATION is more challenging to compute. EXPLOITATION provides the fastest convergence within the first 100 epochs. RANDOM converges in approximately 250 epochs. UPDATE does not converge before the 400th epoch as it does not prefer *aggregators* from the unexploited view and is influenced by the changes of the selected states.

Figure 6 illustrates the messages $\lambda(sessions)$ sent during the aggregation sessions. Under synchronous changes, the distribution of the communication cost during runtime depicted in Figure 6a corresponds to the exploitation and update outcomes in Figure 6a and 6b respectively. EXPLOITATION and UPDATE minimize the messages exchanged to 0 when the aggregates converge to their actual values. This is not the case for RANDOM that continuously exchanges $\lambda(sessions) = nz3 = 1500 * 10 * 3 = 45000$ messages during runtime. These messages are generated by 1500 nodes that periodically establish 10 bidirectional aggregation sessions with 3 messages exchanged in each session. This is the how

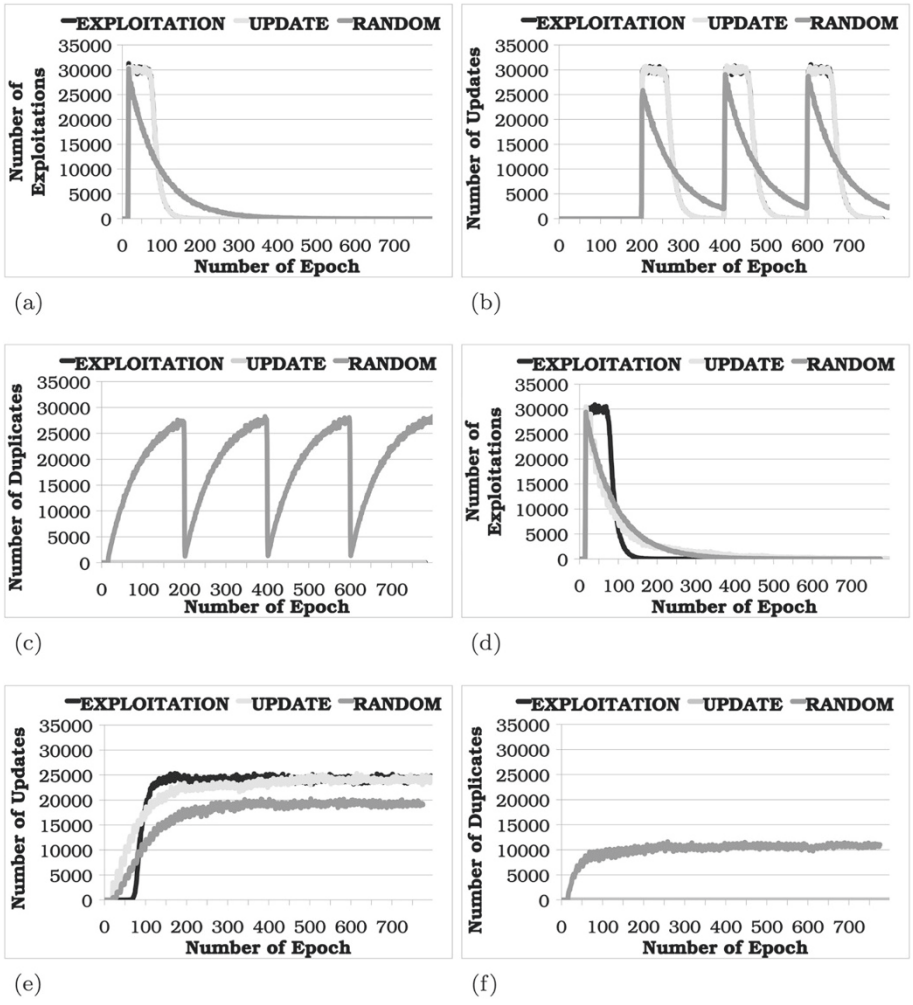

Figure 5 Outcomes of adaptation strategies. (a) Exploitation outcomes under synchronous changes. **(b)** Update outcomes under synchronous changes. **(c)** Duplicate outcomes under synchronous changes. **(d)** Exploitation outcomes under asynchronous changes. **(e)** Update outcomes under asynchronous changes. **(f)** Duplicate outcomes under asynchronous changes.

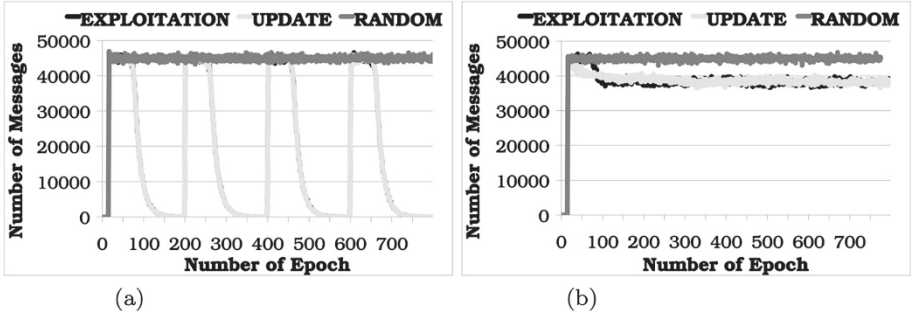

Figure 6 Communication cost of adaptation strategies. (a) Synchronous changes. **(b)** Asynchronous changes.

the communication cost is estimated for larger networks or a different frequency of aggregation sessions. Under asynchronous changes and during convergence in the first 100 epochs, EXPLOITATION and UPDATE exchange the maximum number of $38000 - 45000$ messages that converges to 38000 in the next epochs during which update outcomes are mainly reached.

This communication cost is significantly lower if the nodes of the network do no run both an *aggregator* and a *disseminator* agent. For example, if the network has 500 of its nodes with an *aggregator* and the rest 1000 nodes with a *disseminator*, the communication cost is computed in this case as $1000 * 10 * 2 = 20000$ messages that is significantly lower than the aforementioned upper communication cost.

Bloom filter aggregation memberships

This section investigates the impact of false positives in the accuracy α of aggregates and the communication cost. Specifically, the bloom filters scheme of m = 16 and k = 24 is compared with two other schemes prone to false positives according to the empirical investigations: (i) m = 14, k = 24 and (ii) m = 14 and k = 6.

Concerning the accuracy of the computed aggregates, no significant influence is observed in the two schemes prone to false positives. The matching α between for both (i) aggregation strategies and (ii) synchronous/asynchronous changes remains almost intact. For example, the bloom filter scheme with m = 14 and k = 24 results in a 0.01 lower matching of AVERAGE under synchronous changes compared to the one with m = 16 and k = 24.

Figure 7 compares the outcomes of aggregation sessions for EXPLOITATION in each bloom filter scheme. The results of UPDATE are similar and, therefore, are omitted. The exploitation outcomes are also omitted as they show similar results to those of Figure 7a and 7d. The bloom filter schemes with m = 14, k = 24 and m = 14, k = 6 miss update outcomes that actually result in inconsistency outcomes. This is shown in Figure 7a and 7b under synchronous changes and in Figure 7d and 7e under asynchronous changes. Figure 7c and 7f show that the scheme with m = 14, k = 24 generates 6 and 9 duplicate outcomes as a result of false positives during classification.

False positives result in a higher number of messages $\lambda(sessions)$. During classification, false positives result in inconsistency outcomes. An *aggregator* A_j is incorrectly classified in the outdated and exploited views if the memberships $M_{\mathsf{D}_i}(\mathsf{A}_j)$ and $M_{S'_i}(\mathsf{A}_j)$ are false positive respectively. Under synchronous changes, incorrectly classified exploited *aggregators* move to the outdated view causing inconsistency outcomes. Figure 8 illustrates the number of exchanged messages during runtime for the three bloom filter schemes. EXPLOITATION is adopted. The results of UPDATE are similar and, therefore, are omitted.

Inconsistency outcomes raise the total number of messages exchanged by 15%. The same holds for asynchronous changes but the effect is much smaller as changes in the selected states occur more frequently. In this case, the increase is 2%.

Comparison with related work

Providing a fair quantitative comparison of DIAS with related mechanisms is challenging as DIAS is designed to be a more generic aggregation service and therefore, it serves a different purpose. Yet, this section illustrates a number of quantitative observations concerning the performance of DIAS in comparison with related methodologies.

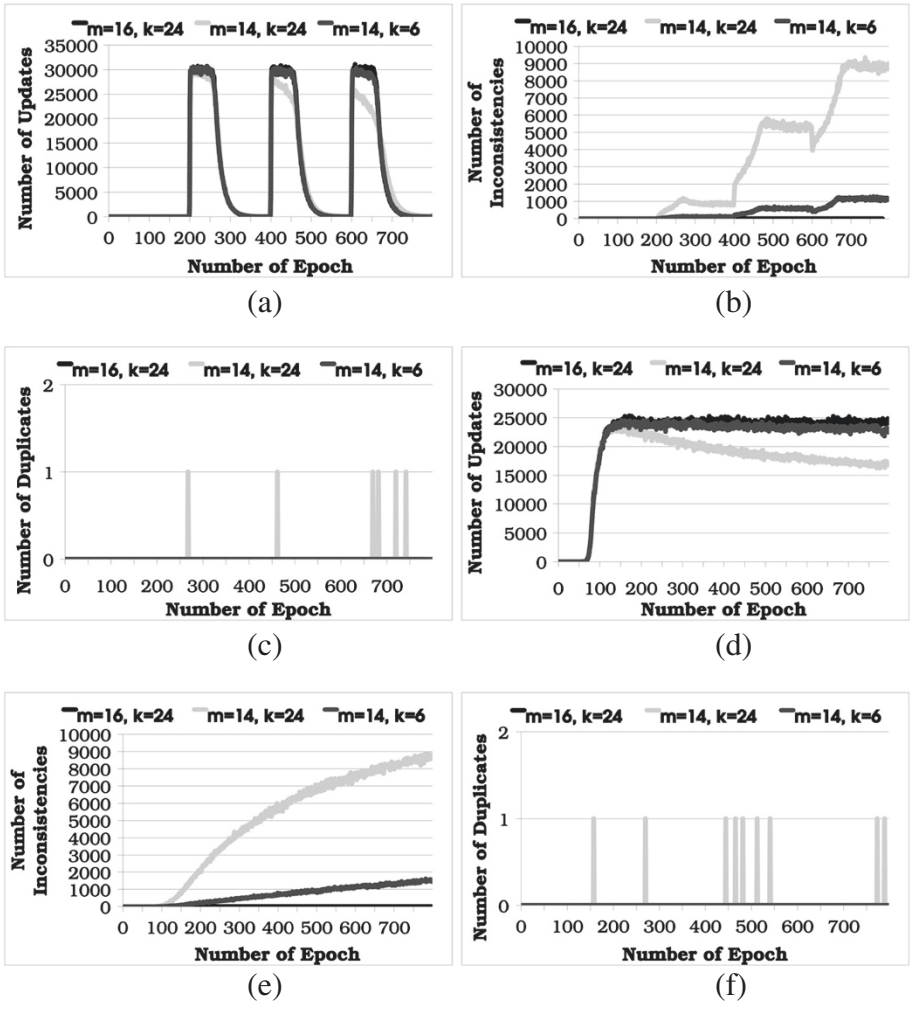

Figure 7 Outcomes of bloom filter schemes. (a) Update outcomes under synchronous changes. **(b)** Inconsistency outcomes under synchronous changes. **(c)** Duplicate outcomes under synchronous changes. **(d)** Update outcomes under asynchronous changes. **(e)** Inconsistency outcomes under asynchronous changes. **(f)** Duplicate outcomes under asynchronous changes.

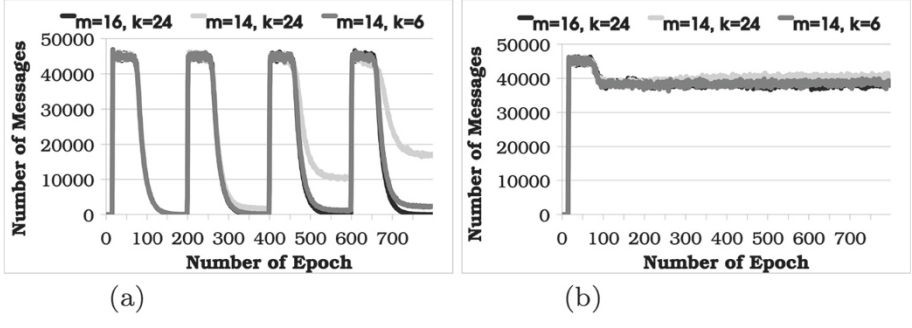

Figure 8 Communication cost of bloom filter schemes. (a) Synchronous changes. **(b)** Asynchronous changes.

For example, gossip-based variance reduction (Jelasity et al. 2005) computes AVERAGE approximately $4-5$ faster than DIAS under static aggregation values. This because the accuracy convergence of DIAS requires approximately 100 epochs, whereas the gossip-based variance reduction converges in $20-25$ (Jelasity et al. 2005) iterations. For synchronous changes, the performance of the two aggregation methodologies, i.e., number of messages and convergence speed, becomes comparable as the iterative variance reduction algorithm requires recomputation of the aggregates. This performance impact becomes more significant as the frequency of changes increases, for example, more than $4-5$ times faster convergence for DIAS. Furthermore, if changes become asynchronous, gossip-based aggregation (Jelasity et al. 2005) becomes infeasible. Recomputations of aggregates cannot be performed as they require some type of synchronization.

Finally, DIAS does not require any changes in its aggregation methodology if different aggregation functions need to be computed simultaneously. This is the most cost-effective use of DIAS that motivates its selection for aggregation over related methodologies.

Diffusion methodologies cannot be applied to a wide range of aggregation functions and are usually interaction-dependent. For example, MAXIMUM and MINIMUM require the communication cost of epidemics (Jelasity et al. 2005; Kashyap et al. 2006) that approaches the one of DIAS.

Other information diffusion and gossiping aggregation mechanisms (Haridasan and van Renesse 2008; Jelasity et al. 2005; Kennedy et al. 2009; Kempe et al. 2003; Nath et al. 2008) do not consider dynamic changes of the aggregation values and assume synchronized recomputations. Coordination of these recomputations in distributed environments in not straightforward. Synopsis diffusion mechanisms (Ahmed et al. 2006; Nath et al. 2008) incorporate incremental updates of aggregates if changes in the aggregation values occur. However, only a relatively low number of changes can be tolerated compared to DIAS. For example, DIAS tolerates in the illustrated experiments 33600 changes compared to 1000 changes (Ahmed et al. 2006). A high number of items in the bit vectors of synopsis diffusion causes significant inaccuracies. The false positives of DIAS do not influence the accuracy of aggregates as they can be detected and eliminated.

Robust tree overlays are a flexible methodology to compute a wide range of aggregates but require topology self-management (Pournaras et al. 2010) in decentralized environments. Communication and storage complexity can be higher than the aggregation itself. Performing a relevant evaluation and comparison of aggregation trees with other more dedicated to aggregation mechanisms, such as DIAS, requires a use-case context and a specific application scenario. If tree topologies are reused between different distributed applications, including aggregation, the allocated cost is shared between these applications something that makes the use of trees more effective Fei et al. 2001. The unique paths of tree topologies are not required in DIAS as unique aggregation values are identified by the classification in the middle level. Furthermore, tree aggregation suffers from an unequal load distribution in nodes and the impact of failures (Ogston and Jarvis 2010). The nodes close to the root receive a high number of forwarded messages from the bottom nodes. Similarly, the impact of a failure close to the leaves is small whereas a single failure close to the root partitions the overlay network. These issues do not concern DIAS as it does not depend on a specific

interaction mechanism. Nonetheless, the realization of the bottom level by the peer sampling service (Jelasity et al. 2007) results in a uniform communication overhead between nodes.

Discussion and future work

The DIAS architecture provides three levels of abstraction and modularity. The top level does not have any knowledge about the underlying complexity of classification and aggregation memberships. A wide range of aggregation functions can be accurately computed as the middle level guarantees that *aggregator* samples are classified as unexploited or outdated. Similarly, the middle level receives *aggregator* samples discovered by the bottom level.

A key feature of DIAS is the predefined number of possibles states during an aggregation phase. A large number of applications are fundamentally based on this assumption and design. User ranking aggregation in recommender systems (Garcin et al. 2009), is based on a finite and often restricted number of options for a user to rank an element. In applications of demand-side energy management (James et al. 2006; Pournaras et al. 2010), aggregate information about a finite number of alternative demand options improve the stability of the Smart Power Grid.

Dissemination and collection of all aggregation values in every agent of the network requires a significant communication cost. One way to decrease this cost is to eliminate the number of *aggregators* and *disseminators* in a network. Section "Adaptation strategies" shows that the communication cost of DIAS is decreased more than half if the network is split into the 2/3 of the nodes running *disseminators* and 1/3 *aggregators*. It is not always necessary for each node to perform both aggregation and dissemination as various applications do not require this. This is especially the case if nodes have different roles in a network, e.g., consumers and producers in the Smart Power Grid.

DIAS is based on the exchange of *aggregator* samples instead of *disseminator* samples. In the current design of DIAS, aggregation values are disseminated to *aggregators* instead of the *aggregators* requesting the aggregation values. The $M_{\mathsf{A}_i}\left(\mathsf{S}'_j\right)$ membership of *aggregators* cannot be used during the classification process as the selected state S'_j is not known. This issue can be overcome by injecting the selected state in *disseminator* samples exchanged by the bottom level.

Experimental evaluation illustrates the high accuracy and matching achieved even in the case of false positives in bloom filters. Tolerance to false positives provides large data space savings. Accuracy is maintained even if the size of bloom filters decreases significantly, resulting in a high number of detected false positives. A future extension is the dynamic and automated allocation of larger space in the bloom filters based on accuracy requirements under false positives. Alternative approaches to bloom filters are also considered in future work, e.g., hash compaction (Dillinger and Manolios 2004).

The classification of *aggregator* samples in the aggregation pool proactively prevents duplicate outcomes that increase communication overhead. The mutual membership checks reactively detect duplicate outcomes not detected during classification due to false positives. Mutual membership checks guarantee highly accurate aggregates, especially in the case of duplicate-sensitive aggregation functions such as SUMMATION, without

introducing additional communication cost. The performance of RANDOM shows the large communication cost that duplicate outcomes cause and the large savings achieved by EXPLOITATION and UPDATE. Other future work concerns the evaluation of DIAS and its applications in various network conditions, such as churn (Kennedy et al. 2009) and latency.

Conclusions

This paper concludes that DIAS is a generic and middleware service for dynamic decentralized aggregation in large-scale distributed networks. The aggregation approach of DIAS is holistic: a local and duplicate-free availability of the distributed aggregation values that enables the simultaneous computation of almost any aggregation function. Achieving this abstraction in a cost-effective manner and without depending on a specific interaction mechanism is a challenge that has not been addressed in related work. DIAS meets these requirements by introducing an implicit representation and storage of the explicit distributed aggregation values: aggregation memberships in bloom filters. Ultimately, the generic design and applicability of DIAS results in a higher communication overhead compared to methodologies based on information diffusion (Jelasity et al. 2005; Nath et al. 2008). This is the trade off end users of such aggregation systems have to deal with: more generic applicability versus higher communication overhead.

The experimental evaluation shows that DIAS achieves high accuracy under synchronous and asynchronous changes of the aggregation values. Even when using bloom filters with a high number of false positives, accuracy is maintained almost entirely due to the mutual membership checks. The classification of *aggregator* samples and their selection based on two adaptation strategies provide (i) the minimization of duplicates that increase inaccuracies and communication overhead and (ii) the intelligent adaptation of aggregation in different network conditions.

Appendix A: Overview of related work

Table 1 summarizes the related aggregation mechanisms discussed in this paper.

Table 1 An overview of related decentralized mechanisms to DIAS

	Aggregation function	Aggregation values	Interaction requirements	Storage requirements
DIAS	any	highly dynamic	dissemination and collection	bloom filters
(Ahmed et al. 2006)	SUMMATION[a], COUNT, AVERAGE, STANDARD, DEVIATION[b]	dynamic	flooding, gossiping or random walks	counting sketches
(Haridasan and van Renesse 2008)	distribution of aggregation values	static	gossiping	synopsis diffusion
(Jelasity et al. 2005)	AVERAGE, COUNT[c], SUMMATION[a]	static, recomputations	gossiping	hash maps for COUNT
(Kashyap et al. 2006)	algorithm variations for MINIMUM, MAXIMUM, SUMMATION, AVERAGE, RANK	static	group formation and gossiping	synopsis diffusion

Table 1 An overview of related decentralized mechanisms to DIAS *(Continued)*

(Kempe et al. 2003)	algorithm variations for SUMMATION, AVERAGE and quantiles	static	gossiping	synopsis diffusion
(Nath et al. 2008)	SUMMATION, COUNT	static	ring/tree topologies, flooding	synopsis diffusion
(Ogston and Jarvis 2010)	SUMMATION[d] queries	dynamic	tree topology	parent and children

[a]It is derived by the AVERAGE and COUNT aggregates.
[b]It is derived by the SUMMATION and its squares.
[c]It is computed using the 'inverse birthday paradox' as explained in Section "Problem description".
[d]Others aggregates could be potentially computed.

Appendix B: Experimental settings

Table 2 summarizes the selected experimental settings. Note that multiple values for a single parameter denote the tested variations of this parameter in some of the illustrated experiments. The values depicted with bold are the default ones.

Table 2 The experimental settings for the evaluation of DIAS

	Parameter	Value
Protopeer	n	1500
	t(DIAS)	800
	T(DIAS)	1000
	t'(DIAS)	6
	$\lvert\mathbf{v}(ring)\rvert$	5
Application	t'(*application*)	15
	v	5
	type of states	numerical
	input domain of states	[0,1)
	generation of possible states	beta distribution
	distribution for s_i^0	alpha=5, beta=25
	distribution for s_i^1	alpha=25, beta=5
	distribution for s_i^2	alpha=10, beta=5
	distribution for s_i^3	alpha=5, beta=10
	distribution for s_i^4	alpha=5, beta=5
	selection of a possible state	cyclical
	T(*application*)	10 (asynchronous), 200 (synchronous)
	P_c(*time*), P_c(*parameter*)	(1.0, 1.0), (0.4, 0.7)
Top Level	T(*top*)	1000
	f()	AVERAGE, SUMMATION, MAXIMUM
Middle Level	T(*middle*)	1000
	z	10
	q	45
	e	15
	adaptation strategy adoption	static
	hashing scheme	double hashing
	m, k	**(16, 24)**, (14, 24), (14, 6)

Table 2 The experimental settings for the evaluation of DIAS *(Continued)*

Bottom Level	r	50
	$T(bottom)$	250
	view selection policy	swapper
	view propagation policy	push-pull
	peer selection	policy random

The bold values are the default ones in the performed experiments.

Appendix C: Source experimental data

Figure 9 shows the five beta distributions[b] used for the parameterization of the possible states in the experimental evaluation of DIAS. In each aggregation epoch, a random value from a given beta distribution is assigned to a possible state.

Figures 10, 11, 12 and 13 illustrate the source data of the experimental results based on which the accuracy and matching are computed.

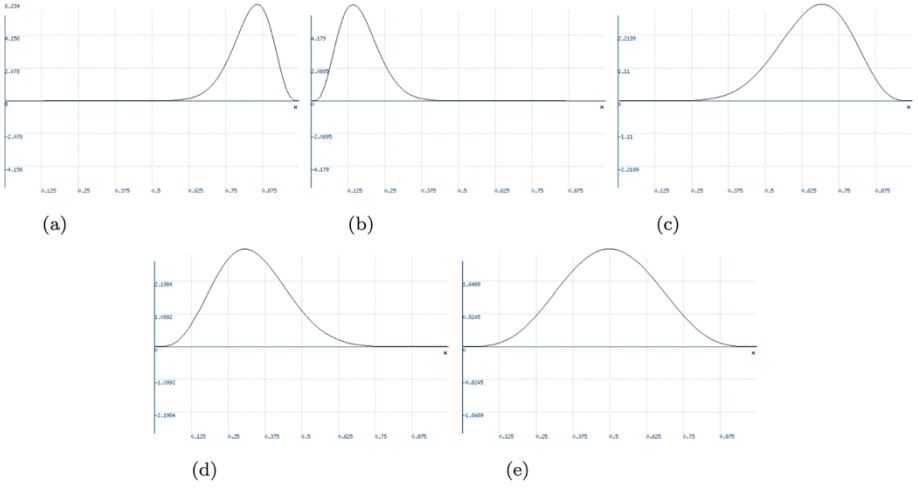

Figure 9 Five beta distribution used as the input domain of the possible states. (a) *alpha* = 25, *beta* = 5. **(b)** *alpha* = 5, *beta* = 25. **(c)** *alpha* = 10, *beta* = 5. **(d)** *alpha* = 5, *beta* = 10. **(e)** *alpha* = 5, *beta* = 5.

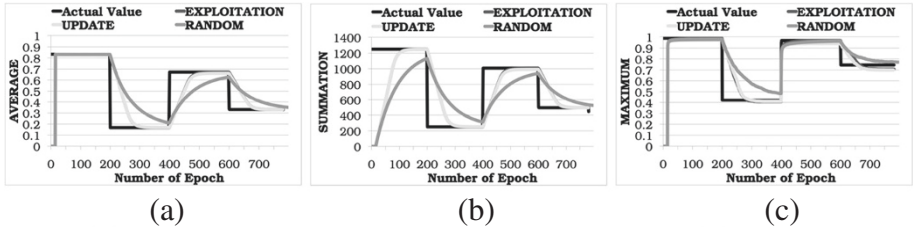

Figure 10 Aggregates of adaptation strategies under synchronous changes. (a) AVERAGE. **(b)** SUMMATION. **(c)** MAXIMUM.

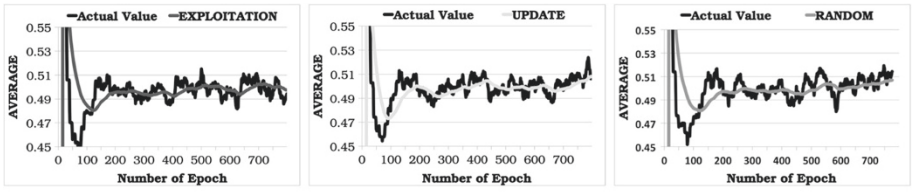

Figure 11 AVERAGE of the adaptation strategies under asynchronous changes. (a) EXPLOITATION. **(b)** UPDATE. **(c)** RANDOM.

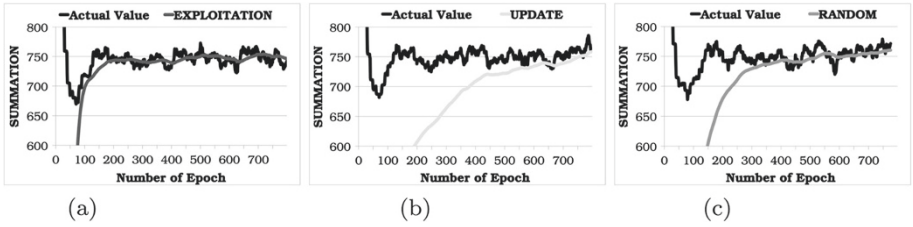

Figure 12 SUMMATION of adaptation strategies under asynchronous changes. (a) EXPLOITATION. **(b)** UPDATE. **(c)** RANDOM.

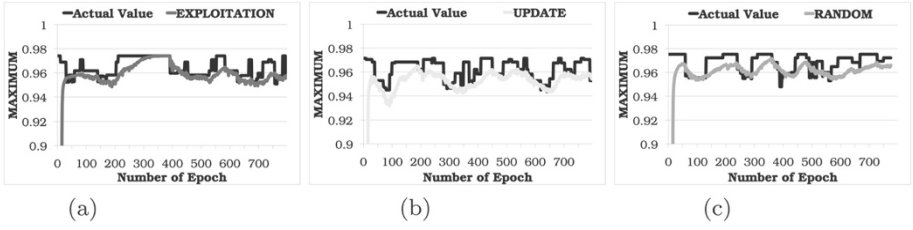

Figure 13 MAXIMUM of adaptation strategies under asynchronous changes. (a) EXPLOITATION. **(b)** UPDATE. **(c)** RANDOM.

Endnotes

[a]During system bootstrapping, there is no need for available historical information to distinguish between different classes as each aggregation value is by default unexploited.

[b]Generated by the Wessa online statistics software, available at: http://www.wessa.net/ (Last accessed: January 2013).

Author details
[1]Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology, Mekelweg 4, 2628 CD, Delft, Netherlands. [2]Faculty of Technology, Policy and Management, Delft University of technology, Jaffalaan 5, 2628 BX Delft, Netherlands.

Received: 12 June 2013 Accepted: 30 August 2013
Published: 08 Nov 2013

References

Ahmed N, Hadaller N, Keshav S: **Incremental maintenance of global aggregates.** 2006. Tech. rep., University of Waterloo, Waterloo, Ontario.

Bloom BH: **Space/time trade-offs in hash coding with allowable errors.** *Commun ACM* 1970, **13**(7):422–426.

Calvo T, Kolesárová A, Komorníková M, Mesiar R: **Aggregation operators: properties, classes and construction methods.** In *Aggregation Operators: New Trends and Applications, Volume 97 of Studies in Fuzziness and Soft Computing,* Heidelberg, Germany: Physica-Verlag GmbH; 2002:3–104.

Chitnis L, Dobra A, Ranka S: **Aggregation methods for large-scale sensor networks.** *ACM Trans Sensor Netw* 2008, **4**(2):1–36.

Deke G, Yunhao L, Xiangyang L, Panlong Y: **False negative problem of counting bloom filter.** *IEEE Trans Knowl Data Eng* 2010, **22**(5):651–664.

Dillinger PC, Manolios P: **Bloom filters in probabilistic verification.** In *Proceedings of the 5th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2004, Volume 3312 of Lecture Notes in Computer Science,* Heidelberg: Springer-Verlag, Berlin; 2004:367–381.

Fei A, Cui J, Gerla M, Faloutsos M: **Aggregated multicast with inter-group tree sharing.** In *Proceedings of the 3rd International Workshop on Networked Group Communication, NGC 2001, Volume 2233 of Lecture Notes in Computer Science,* Heidelberg: Springer-Verlag Berlin; 2001:172–188.

Galuba W, Aberer K, Despotovic Z, Kellerer W: **ProtoPeer: a P2P toolkit bridging the gap between simulation and live deployement.** In *Proceedings of the Second International Conference on Simulation Tools and Techniques, ICST 2009 Gent,* Belgium: ACM; 2009:1–9.

Garcin F, Faltings B, Jurca R, Joswig N: **Rating aggregation in collaborative filtering systems.** In *Proceedings of the 3rd ACM Conference on Recommender Systems, RecSys 2009,* New York, NY, USA: ACM Press; 2009:349–352.

Haridasan M, van Renesse R: **Gossip-based distribution estimation in peer-to-peer networks.** In *Proceedings of the 7th International Workshop on Peer-to-peer Systems, IPTPS 2008.* Berkeley, CA, USA: USENIX Association; 2008:13.

James G, Cohen D, Dodier R, Platt G, Palmer D: **A deployed multi-agent framework for distributed energy applications.** In *Proceedings of the 5th International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2006,* New York, NY, USA: ACM Press; 2006:676–678.

Jelasity M, Montresor A, Babaoglu O: **Gossip-based aggregation in large dynamic networks.** *ACM Trans Comp Syst* 2005, **23**(3):219–252.

Jelasity M, Voulgaris S, Guerraoui R, Kermarrec AM, van Steen M: **Gossip-based peer sampling.** *ACM Trans Comp Syst* 2007, **25**(3).

Jerzak Z, Fetzer C: **Bloom filter based routing for content-based publish/subscribe.** In *Proceedings of the 2nd International Conference on Distributed Event-based Systems, DEBS 2008,* New York, NY, USA: ACM Press; 2008:71–81.

Jiang S, Guo L, Zhang X: **LightFlood: an efficient flooding scheme for file search in unstructured peer-to-peer systems.** In *Proceedings of the 2003 International Conference on Parallel Processing, ICPP 2003,* Los Alamitos, CA, USA: IEEE; 2003:627–635.

Gkantsidis C, Mihail M, Saberi S: **Random walks in peer-to-peer networks: algorithms and evaluation.** *Perform Eval* 2006, **63**(3):241–263.

Kashyap S, Deb S, Naidu KVM, Rastogi R, Srinivasan A: **Efficient gossip-based aggregate computation.** In *Proceedings of the 25th Symposium on Principles of Database Systems - PODS 2006,* New York, NY, USA: ACM Press; 2006:308–317.

Kennedy O, Koch C, Demers A: **Dynamic approaches to in-network aggregation.** In *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009.* Los Alamitos, CA, USA: IEEE; 2009:1331–1334.

Kempe D, Dobra A, Gehrke J: **Gossip-based computation of aggregate information.** In *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2003,* Washington, DC, USA: IEEE Computer Society; 2003:482–491.

Li F, Pei C, Jussara A, Andrei BZ: **Summary cache: a scalable wide-area web cache sharing protocol.** *IEEE/ACM Trans Netw* 2000, **8**(3):281–293.

Nath S, Gibbons PB, Seshan S, Anderson Z: **Synopsis diffusion for robust aggregation in sensor networks.** *ACM Trans Sensor Netw* 2008, **4**(2):1–40.

Ogston E, Jarvis SA: **Peer-to-peer aggregation techniques dissected.** *Int J Parallel, Emergent and Distributed Syst* 2010, **25:**51–71.

Pournaras E: **Multi-level reconfigurable self-organization in overlay services.** *PhD thesis.* Delft University of Technology, Netherlands 2013.

Pournaras E, Warnier M, Brazier FMT: **Adaptation strategies for self-management of tree overlay networks.** In *Proceedings of the 11th IEEE/ACM International Conference on Grid Computing, Grid 2010.* Los Alamitos, CA, USA: IEEE; 2010:401–409.

Pournaras E, Warnier M, Brazier FMT: **Local agent-based self-stabilisation in global resource utilisation.** *Int J Auton Comput* 2010, **1**(4):350–373.

Yuh-Jzer J, Chien-Tse F, Li-Wei Y: **Keyword search in DHT-based peer-to-peer networks.** In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems, ICDCS 2005.* Los Alamitos, CA, USA: IEEE; 2005:339–348.

10.1186/2194-3206-1-19
Cite this article as: Pournaras *et al.*: A generic and adaptive aggregation service for large-scale decentralized networks. *Complex Adaptive Systems Modeling* 2013, 1:19