September 2018

EDEN: Extensible Discussion Entity Network[1]

Christian METER, Alexander SCHNEIDER [2] and Martin MAUVE
Computer Networks Department
Heinrich-Heine University Düsseldorf, Germany
firstname.lastname@hhu.de

Abstract. Enabling the reuse of arguments as entities that can be shared across multiple Internet-based discussion platforms and that can be improved upon while they are being used and reused has many benefits ranging from easier participation in an online discussion to increasing the quality of arguments. In this paper we propose a mechanism that is able to support the large-scale reuse of arguments by providing distributed version control of argument data. Building on that mechanism we have designed and implemented *EDEN*, a framework which enables platform providers to easily network their discussions. EDEN is designed for real-world use and provides all tools necessary to enable the reuse of arguments and their interrelation for users and providers alike.

Keywords. massive online discussion, discussion networks, EDEN, discussion graphs

1. Introduction

Arguments and their interrelation are valuable resources. They require effort to craft and they reflect the knowledge and opinions of those that have contributed them. Furthermore, their value grows as a network of arguments and their interrelations increases in size. On the Internet this is currently not supported in an appropriate way. Most arguments are ephemeral postings in forums and comment sections of news media. Even dedicated argumentation websites do not allow for connecting arguments across multiple websites. In order to address this problem Bex, Lawrence, Snaith and Reed have introduced the notion of an *Argument Web* [6]. Unfortunately, the Argument Web has not (yet) gained sufficient traction and is limited to a set of research prototypes.

In this paper we argue that in order to have a larger impact, the Argument Web needs to be more than a way to specify, describe and reference arguments. In particular it has to take into account the specific needs of those that operate websites where argumentation is taking place and of the users that visit those web sites. As we shall discuss, this leads to a challenge on the system level that can be summarized in a simple question: How can arguments and their interrelations be managed as persistent resources in a distributed

[1]Submitted to IAT.
[2]Both first authors contributed in equal parts to this work.

(web-based) system? As an answer to this question we propose the idea of an *Extensible Discussion Entity Network* (EDEN).

EDEN is designed to provide persistent arguments and interrelations between them, which can be shared and reused, while incorporating the manifold requirements of users and platform providers. Those, sometimes contradicting, needs are usually not considered, when designing systems in the argumentation space. We believe that EDEN facilitates adoption for real-world scenarios.

The goals of this paper are twofold. First we would like to raise awareness for the fact that there are system-level challenges that need to be addressed in order to make the idea of an Argument Web work in a real-world setting. Second, we present a solution for the most important of those challenges, namely how to distribute and manage argument data in an heterogeneous Internet-based environment.

The remainder of the paper is structured as follows. Section 2 examines related work and compares it to our contribution. Next, we introduce our view on distributed argumentation and its stakeholders in Section 3. Section 4 then discusses a method for versioning arguments in a distributed environment. Following that, we present an implementation of EDEN which describes its functionality and specifics in Section 5. Finally, in Section 6 we conclude the paper with a summary and an outlook on future work.

2. Related Work

The idea of a connected network of arguments is not entirely new. The general idea for an "Argument Web" was established by Rahwan et al. [11] and further refined by Bex et al. [6]. Following the general idea a central database for the Argument Web was created by Lawrence et al. [8] which in turn interoperated with different applications belonging to the Argument Web [5,7]. The point where EDEN differs from that work is that we do not utilize a central database, which acts as a central interface for import and export for arguments in the AIF format. Instead we aim for dynamic exchange in a federated network of providers. Furthermore, EDEN is not bound to any special ontology, but instead focuses on arbitrary "atomic" entities.

There is also work by Rowe et al. [12] where the concept of reuse is anticipated by designing a system where it is possible to import and export arguments into and out of the Araucaria system on local instances.

Argument reuse has also been touched upon outside of the argumentation community. Kelly et al. [14] proposed the reuse of arguments via design patterns to ease the construction of safety cases and Smith and Harrison [13] proposed a system for reuse of descriptive arguments in hazard classification. To our knowledge EDEN is the first system to aim for reuse of arguments made by layman in a distributed online argumentation environment.

3. Reusable Arguments and their Environment

In order to be able to tackle the systems-level challenges posed by the idea of reusable arguments, we need a good understanding of the environment, where those arguments are created and (re)used. This environment consists of websites and web-based services that host discussions. In particular, this includes online newsmedia, social networks and discussion forums. We term these websites and web-based services *argument aggregators*, since they aggregate arguments provided by users in order to form online discussions.

Argument aggregators typically have policies on what an acceptable user-provided argument is and they have mechanisms in place that ensure that the contributions of the users adhere to those policies. The policies of different argument aggregators are quite heterogeneous, thus the same arguments might be acceptable for some content aggregators while others would consider them a violation of their policies. Furthermore, argument aggregators typically perceive the arguments provided by the users as a valuable commodity which helps gain page impressions and generate income, hence they are unlikely to be willing to share them, unless they get something in return like a reference to their web-site or something similar.

Arguments consist of statements that are linked to each other by different types of relations. They are regularly provided by the users of an argument aggregator. Arguments are also often linked to the content of the argument aggregator, e.g. they might pertain to a discussion regarding a blog entry or a news-media article. A specific argument is initially submitted by a single user to a single argument aggregator. However, any user might later on be willing to improve the argument, for example by correcting spelling errors or by making a statement more concise. The users might also want to use a given argument in another discussion, potentially hosted by a different argument aggregator.

Arguments are interconnected. Each argument, potentially, has numerous relations to other arguments. Furthermore, arguments might only be valid in a specific context. I.e., an argument might contain implicit information, that are not specifically stated. For example, the argument “Our labs are in bad shape, therefore we need to invest in new lab equipment.” includes implicit information about the condition of the author’s working environment since not all existing laboratories are in bad shape.

4. Distributed Management of Arguments

The characteristics of argument aggregators, users and arguments lead to challenges at the systems level that need to be addressed in order for the idea of persistent and reusable arguments to come true. The most prominent one is the development of a suitable architecture for the storage and distribution of arguments, where arguments are updated in an appropriate way, if they are used by multiple argument aggregators.

Since argument aggregators are independent entities that desire autonomous control over the arguments they store, show to their users and distribute to other argument aggregators, the architecture of a system for reusable and persistent arguments needs to be distributed. Given that arguments and their interrelations can be modified and improved upon over time, this immediately raises the question how their shared state can be managed.

One option is to take all proposed updates and calculate a resulting state that is then used by every argument aggregator. This, however, entails two problems. First, there needs to be a mechanism calculating a shared global state, which is a hard, but potentially solvable, problem in a distributed system. Second, all argument aggregators would have to agree unanimously on how to handle all updates – in particular whether to accept a given update or reject it. This is unlikely to be feasible in a real world environment.

If updates are optional, however, arguments may have different states at different aggregators. This inconsistent state is likely to cause problems. For example, an attack on an argument may be valid only for a certain variant of that argument that exists only on a subset of providers, since others modified it. Thus it is not clear how the attack can be reused in a distributed environment where aggregators have different versions of the attacked argument.

To solve this dilemma, we propose an approach derived from distributed source code versioning: arguments, or rather the statements and interrelations that make up the arguments, do have a version. An update produces a new version without modifying the original. The updated version refers to the original(s) as it's predecessor(s), effectively preserving history. This allows both for persistence, since no version is ever deleted and free choice of the content aggregators regarding what updates to accept.

In order to support distributed versioning of arguments, two problems have to be addressed. On the one hand, appropriate data structures are required that support the versioning of arguments. On the other hand there needs to be a mechanism to distribute information about new versions to those that might be interested in updates.

4.1. Data Structures for Versioning Arguments

In order to provide versioning for arguments we first determine the entities that make up a network of arguments. Those are statements and relations between statements. We then define an object to be a specific version of a specific entity. The data structures for storing objects have some common elements for both statement objects and relation objects: a global identifier $\mathscr{N}_{host}$ for the argument aggregator that created the current object (for example, the DNS host name of the argument aggregator), a local identifier $\mathscr{N}_{id}$, that uniquely identifies the entity stored in the object amongst all the entities that this argument aggregator has created objects for, and a version number $\mathscr{N}_{version}$ that indicates a specific version of the entity at this argument aggregator. Together those three values represent the object-id $\mathscr{N}$ which uniquely identifies a specific object. An important aspect of the object-id is that it can be determined locally and does not have to be coordinated amongst argument aggregators.

Furthermore, each object also has a flag d that indicates if it has been marked for deactivation. The latter is required since nothing should ever truly be deleted when doing versioning. Therefore a deletion of an object is just signaled by a specific version of that object where this flag is set. Providers can chose to follow the deactivation by making the object inaccessible to their users.

In addition to the information that is common to all objects, a statement object contains the following information:

- $\mathscr{P}$: a set of pointers to immediate predecessor versions, which is either a set of object-ids, or $\mathscr{P} = \emptyset$.
- $\mathscr{C}$: the data that makes up the statement, typically a plain text and meta information such as the author of the statement and the authors of modifications to the statement.

Summarizing, a statement-object can be fully described as a tuple $\langle \mathscr{P}, \mathscr{N}, \mathscr{C}, d \rangle$.

Relations include the following additional information:

$\mathscr{S}$: the relation's source, which is the object-id of any statement

$\mathscr{D}$: the destination of the relation, which is the object-id of any statement or another relation

t : the relation-type, e.g. "attack" or "premise-conclusion-relation"

A relation-object is thus described by: $\langle \mathscr{N}, \mathscr{S}, \mathscr{D}, t, d \rangle$. Relations are treated as immutable, they can only be created and deleted, but their content never changes. Therefore they do not need a predecessor.

We do believe that these data structures are sufficiently generic to capture arbitrary argumentation schemes, by utilizing $\mathscr{C}$ as a store for atomic entities of a scheme, and yet they provide all the information required to support versioning. They are also quite easy to extend if the need should arise.

4.2. Versioning Arguments in a Distributed Environment

An object (and thus a specific version of a specific entity) has an *authoritative argument aggregator*. This is the argument aggregator that created it and it can be easily determined by looking at $\mathscr{N}_{host}$ of that object. Another argument aggregator can import that object in order to integrate it into an argumentation that it hosts. After a provider imports an object, it can register with the authoritative aggregator for that object in order to receive updates regarding the entity contained in the object.

When an authoritative content aggregator updates an entity, it creates a new object for the new version of that entity with a new version number. It then notifies the argument aggregators that have registered with it regarding that entity. Those argument aggregators can then choose to accept the update or they can stick with the old version. This is a local decision that could be made by a dedicated moderator, the users of the argument aggregator or by means of a policy where one argument aggregator decides to trust another argument aggregator to provide reasonable updates.

If an entity is updated by an aggregator that is non-authoritative, a *fork* is created. A fork is a new object. For example if the original statement object was $St_1 = \langle \emptyset, id_x, \mathscr{C}, 0 \rangle$ with $id_x = \langle someaggregator.com, 42, 0 \rangle$, then the new fork-object including the new version of that entity could be $St_2 = \langle \{id_x\}, id_y, \mathscr{C}_2, 0 \rangle$ with $id_y = \langle anotheraggregator.org, 13, 0 \rangle$. The aggregator which created the fork is authoritative for that fork. When a fork of a statement is created, all relations belonging to the forked statement are copied and all instances of the forked statement are replaced by the fork in the copied statements. This does not update existing relations, but rather produce new ones specifically for the fork-object.

When an aggregator F creates a fork, it contacts the authoritative aggregator A of the object that was forked. A can decide to ignore the update. Then nothing happens and A remains authoritative for the original object while F is authoritative for the forked object. Or A can accept the update. In that case, it creates a new version of that entity by creating an appropriate object, which has an incremented version-number, updated content and the fork-object as its predecessor, to keep the version history accurate. As with all updates, the new object is then transmitted to all argument aggregators that have registered with the authoritative aggregator regarding that entity. In particular this is received by F. Once F realizes that its update has been accepted, it replaces the fork with the received update.

4.3. Example of Fork and Update Processes

In order to illustrate how the proposed versioning scheme works, we now present an example showcasing the fork and update processes. The example begins as an aggregator with the global identifier *a.com* creates a statement which looks as follows: $S = \langle \emptyset, id_a = \langle a.com, 24, 0\rangle, \mathscr{C}_1, 0\rangle$. Now there are several cases that can occur.

4.3.1. Updating the Statement

Through a user-driven process, *a.com* decides to update the content of the statement S, producing new content $\mathscr{C}_2$. As a consequence an official updated statement-object $\langle \{id_a\}, \langle a.com, 24, 1\rangle, \mathscr{C}_2, 0\rangle$ is created and published to all other aggregators using S. Those aggregators decide individually whether they stick with the old version or update to the new one.

4.3.2. Creating a Fork

An aggregator *b.org* is using S and wants to update the statement's content to $\mathscr{C}_3$. A fork is now created which looks as follows: $\langle \{id_a\}, id_b = \langle b.org, 40, 0\rangle, \mathscr{C}_3, 0\rangle$. This fork is reported to the original aggregator *a.com*. In case *a.com* rejects the update, nothing more happens. If *a.com* accepts the update, it creates an updated version of S and sets the fork as a predecessor to preserve history – resulting in: $\langle \{id_b\}, \langle a.com, 24, 1\rangle, \mathscr{C}_3, 0\rangle$. This is then published to all other aggregators using S. Upon receiving the new object, *b.org* replaces the fork with the update, since its own changes have now been incorporated by *a.com*.

4.3.3. Simultaneous Forks and Updates

Continuing the example in Section 4.3.2, *c.net* is now also using S. It, too, has created an update to S with the content $\mathscr{C}_4$, which results in the object: $\langle \{id_a\}, id_c = \langle c.net, 1337, 0\rangle, \mathscr{C}_4, 0\rangle$. This fork is also communicated to *a.com*, which already updated S after accepting the fork from *b.org*. *a.com* can now choose to incorporate both forks in a new update where the content is then $\mathscr{C}_5$, thus producing $\langle \{id_b, id_c\}, \langle a.com, 24, 2\rangle, \mathscr{C}_5, 0\rangle$. In this version, both the objects from *b.org* and *c.net* are predecessors of the updated object. Figure 1 showcases the relations in this scenario. *a.com* could have also chosen to solely use the fork from *c.net* as the most current version 2, disregarding the changes of *b.org* included in version 1, effectively creating: $\langle \{id_c\}, \langle a.com, st_1, 2\rangle, \mathscr{C}_4, 0\rangle$. Again, this is published to all other aggregators using S. Upon receiving this, *b.org* and *c.net* decide whether they want to stick with their current version or update to the new one.

5. EDEN

This section introduces EDEN, the implementation of the aforementioned ideas. We briefly describe the basic concepts of EDEN before we lay out the modular architecture, several optimizations and first experiences of usage.

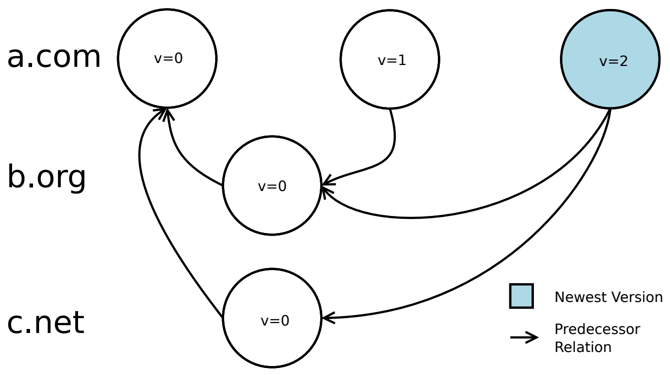

Figure 1. A visualisation of predecessor-relations between different forks and updates of a statement.

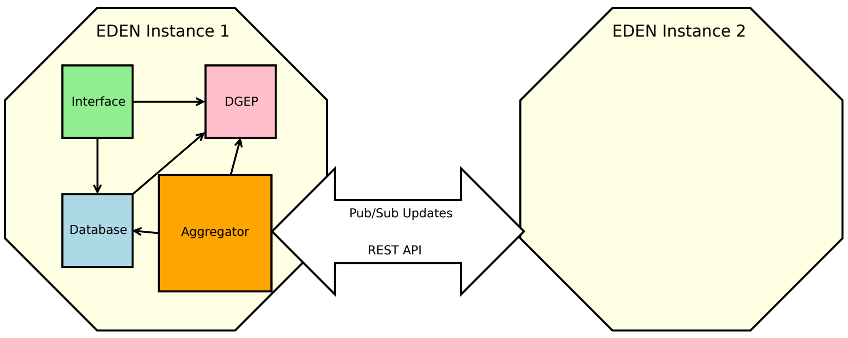

Figure 2. Dataflow between the modules of one EDEN instance. The dataflow with other EDEN instances is established via Pub/Sub and a REST API.

5.1. Basic Concepts

As described in the section above, EDEN is realized as a *federated network* of argument aggregators, where each aggregator is responsible for the state of its own data. Every argument aggregator that wants to enable its community to participate in the global argumentation network, can start up an EDEN instance, which discovers other instances through its initial whitelist and through foreign arguments discovered from those whitelisted instances. The most important task of EDEN is the management and exchange of local and foreign statements and relations. To this end the federated network maintained by EDEN has two logical layers – the local community of an aggregator and the global community spanning all available EDEN instances and their users.

Ideally, EDEN instances should be run by entities which are trusted by their users, like newspaper outlets, NGOs or other organizations. We do not, however, place any firm restrictions on which entities can run an EDEN instance.

We have developed EDEN with modularity in mind. EDEN therefore consists of independent modules, which can be exchanged, as long as they adhere to interface definitions between module “seams”. Everything from the aggregator logic, the interface, the database to the execution logic can easily be customized and exchanged in individual EDEN deployments.

5.2. Architecture

The general architecture and dataflow of EDEN's architecture is shown in Figure 2. There are four main modules at work – each with its own purpose.

The interface module enables layman-users to participate in a discussion with their arguments. This in itself is a non-trivial challenge. We use *discuss*, described in Meter et al. [2], as an example implementation of the interface module. In order to allow users to easily import arguments, we present the user with similar arguments from the local and global community, while the user is trying to formulate their own thoughts into an argument by typing parts of it. Similarity here being the analogous and logical proximity of words being typed in respect to potential new arguments. There are many other ways how this support could be realized, e.g. by being able to bookmark arguments at one argument aggregator and then later on reuse these bookmarks in other argumentations at the same or a different argument aggregator. We chose this method to not impose any extra strain on the user in order to not deter them from using the system.

The aggregator module is, metaphorically speaking, the communication central and brain of the operation. All entities at one point pass through the aggregator module. Its duty can be divided into two sections. First, obtaining data from external EDEN instances and providing the local data back to them. Second, coordinating the internal flow of data to make sure it proceeds efficiently between the modules. Our implementation of the aggregator module provides a REST API to enable foreign EDEN instances to query it for data. We furthermore use the *RabbitMQ* publish/subscribe system for queues, to which the aggregators subscribe to be informed about updates to the subscribed entities.

The database module needs to store and efficiently provide heterogeneous data to the other modules. One could use traditional relational databases, but to simplify the storage and query of potentially big amounts of different data-types, EDEN uses an *Elasticsearch* database. One of the many advantages of Elasticsearch is the semantic search, which allows for sophisticated queries, e.g. searching for synonyms. This helps with the provisioning of relevant arguments in respect to the users input.

Finally there is a *Dialogue Game Execution Platform* (DGEP) as defined by Bex et. al. [4]. We use Krauthoff's *Dialog-Based Argumentation System* (D-BAS) [3] for this purpose. The DGEP is responsible for handling all necessary steps in a discussion, utilizing a predefined set of rules applying to a "natural" discourse. Through the modularity any DGEP could replace D-BAS inside the EDEN framework as long as it adheres to the interface conventions between the modules. Currently, the DGEP module also doubles as the module which creates structure data from user input. The choice for using D-BAS in the default version of EDEN is not made because of any architecture considerations, but because we simply needed to pick any one DGEP we could work with to provide a functioning implementation.

The communication with foreign EDEN instances is established in two different ways. If one instance is looking for an entity which may be stored at a different instance, it can query the remote aggregator via a REST API. This will provide it either with a "not-found" answer in case the entity could not be found or with the found entity and a publish/subscribe channel in the successful case. The querying instance can subscribe to the channel if desired to receive updates about new entities or changes in entities, i.e. new versions, thus making the pub/sub system responsible for push-based updates and the REST API for initial queries and pull-based updates.

5.3. Statements and Links

EDEN uses the object types statement and relation[3] as described above. *Statements* are implemented as shown in Listing 1 with some required and some optional keys. The triplet of `[:aggregate-id, :entity-id, :version]` provides a unique address for a specific version of a statement entity. In particular this address can be used by non authoritative argument aggregators to refer to this version.

```
(s/def ::statement
  (s/keys :req [::author ::content ::created
                ::aggregate-id ::entity-id ::version]
          :opt [::ancestor-aggregate-id ::ancestor-entity-id
                ::ancestor-version]))
```

Listing 1: Definition of a statement.

Links are represented as immutable objects, which are defined by a type, source and destination in our implementation. The type represents the relation (e.g. attack, support, undermine, . . .) and source and destination are references to objects in a specific version[4]. Since the links are immutable, they can be propagated alongside statements through the pub/sub channels and REST API. The aggregators can then resolve the link-references to the statements and show the users the appropriate versions[5].

```
(s/def ::link
  (s/keys
    :req [::author ::type ::created
          ::from-aggregate-id ::from-entity-id ::from-version
          ::to-aggregate-id ::to-entity-id ::to-version
          ::aggregate-id ::entity-id]))
```

Listing 2: Definition of a link.

5.4. Context Dependent Arguments

To properly import an argument into a foreign discussion, the reused data must be context-free. Our initial approach was to reuse statements and links in a way, that automatically included the reuse of all connected links and statements (e.g. attacks and supports) thus linking both argumentation graphs automatically. This does not always work, since statements may implicitly carry context pertaining to a specific discussion. For example if a family is discussing the acquisition of a pet the statement S_1: “Dogs are good family pets” may be used, with the corresponding attack A_1: “We do not have time to

[3]In the implementation relations were called links.

[4]The source is always a statement, while destination can be a link or statement.

[5]The current published version has the destination-version as an optional part for a link. This will change, according to the description in Section 4, in the next release.

walk a dog every day". The attack is true in the context of the family discussion, because it implicitly carries the information, that the family is too busy to care for a dog. If S_1 is now reused in the discussion of an animal-fan forum where the participants want to dedicate a lot of time to their pets and A_1 is automatically presented as an attack, it might not make a lot of sense.

There are different approaches which can be taken to solve this problem. The solution we choose to implement is an "intelligence of the masses" approach. This provides users with the ability to judge about context dependence of automatically imported statements in a review system, before they are fully added and presented to all other users in the discussion. The arguments can be judged one-by-one ordered in a queue accessible to the community members. This works as follows: When a user imports a statement, all other statements which have a relation with it are placed in this new queue. The reviewing users are presented with the statement at the head of the queue, which may be imported if its context-free, as well as with the statement that caused the import of the statement to be judged. The users can then vote to reject or to accept the import. Please note, that the users do not vote on their opinion regarding the content of a statement, but whether the import of it is sensible in the context of the discussion. If a majority of voters accept the import, the statement is fully added to the local discussion and its immediately related statements are placed in the queue. To not overflow the queue with a growing number of review cases, it is capped to a reasonable maximum number of review cases. If the queue is nearing its maximum, statements which are closest to manually imported ones are prioritized. This should prevent the case where one imported statement fills the queue solely with its related statements, while others are left out. The success of this procedure relies on the user's ability to make objective contributions regarding natural language arguments, which is a feasible assumption as shown in a field study [1] for the D-BAS system, where the users were quite capable in reviewing different aspects of reported statements and arguments. A similar approach to include the community is also heavily used on the StackExchange platforms, e.g. *StackOverflow*[6].

5.5. Further Optimizations

We also implemented some optimizations which help EDEN to better perform its tasks of fostering argument reuse.

We implemented a background entity crawler to optimize argument recommendations to the user. The crawler activates periodically when the instance has unallocated resources and queries foreign instances for yet unknown entities which are then indexed to enhance the lookup-time in the future. The crawler always tries to index the most relevant entities first. In our case this means e.g. statements which are directly – and if none can be found – indirectly related to already known statements. This is done because the chances are higher a user will import statements more closely related to statements already present in the discussion than otherwise. Random entities are queried when all related ones are already indexed.

The aggregator, furthermore, uses a tiered system for retrieval of entities to optimize the information-flow. If it is queried for an entity, the aggregator first attempts a lookup inside its cache. Upon failing to find the desired item in the cache, the lookup is directed to the database. If the entity can not be found in the local database either, it is retrieved

[6]`https://stackoverflow.com`

from a foreign EDEN instance. This guarantees that the entities are found as fast as possible, since slower queries to the database and to foreign instances are reduced. Of course the last tier of querying remote aggregators is omitted if the query originated from a foreign instance.

5.6. Hands-On Experience

EDEN was written entirely in Clojure and can be freely obtained at github.com[7]. It can be run without further installation from the *Docker* virtual environment, for which we provide the proper configuration. The Docker container also includes a D-BAS and a discuss instance, which are used as DGEP and interface of EDEN, as mentioned in previous Sections.

We conducted first small-scale tests between two and three instances in small mockup-environments running in different Docker containers. Each container was configured to simulate a physical instance on the same network and we used statements and links which were gathered in a field study using D-BAS [1] and split them up into different subsets used by distinct test-instances.

The tests were not meant as definitive performance simulations or a scientific study, but to get an inkling of how multiple EDEN instances behave together. As we expected, the exchange of arguments worked without any further complications and felt natural to the user. Overall the user-experience did not differ from a normal usage of discuss without the EDEN network – except for the larger selection of pre-formulated arguments – which is a positive sign that the user-facing parts are working as intended and do not inherently add any extra strain on the user. Naturally, this was only conducted to gather a general first experience and we will conduct further real-world tests in the future to obtain more scientifically robust data.

6. Conclusion and Future Work

In this paper we introduced EDEN as a framework to enable discussion-entity reuse between different argumentation platforms. We discussed the challenge of keeping a consistent state in a distributed environment and the resulting challenges for versioning arguments. Our work contains solutions for versioning arguments in a distributed network as well as a solution for context-dependence of entities. Furthermore, we introduced a working implementation of the EDEN framework which is open source and freely available to use. The implementation also contains several technical optimizations and performed successfully in first small-scale tests.

One main challenge that remains as future work is the deployment and evaluation of EDEN by real-world argument aggregators. We are currently in the process of negotiating with companies that provide software for online-participation processes such as participatory budgeting and urban planning. We do believe that this might be an excellent starting point for sharing arguments, since there are many distinct online-participation processes that share common topics. Real world adoption could also be furthered by adding DGEP modules for argument aggregation services like www.debatepedia.org or

[7]https://github.com/hhucn/eden

www.procon.org. or by incorporating argument mining modules for unstructured natural language arguments from e.g. social media.

We also plan to release improved versions of EDEN. Improvements can be pursued by designing methods to ease the reuse of arguments for the users even further. A shared user-base between different EDEN instances could be pursued to facilitate adoption of the network. Additionally, the technical performance of the framework can be improved upon as well.

References

[1] Krauthoff, T., Meter, C., & Mauve, M. (2017). *Dialog-Based Online Argumentation: Findings from a Field Experiment.* Proceedings of the 1st Workshop on Advances In Argumentation In Artificial Intelligence co-located with XVI International Conference of the Italian Association for Artificial Intelligence. Bari, Italy.

[2] Meter, C., Krauthoff, T., & Mauve, M. (2017, July). *discuss: Embedding Dialog-Based Discussions into Websites*. In International Conference on Learning and Collaboration Technologies (pp. 449-460). Springer, Cham.

[3] Krauthoff, T., Meter, C., Betz, G., Baurmann, M. & Mauve, M. (2018, September). *D-BAS – A Dialog-Based Online Argumentation System*. Computational Models of Argument (pp. 325-336), Warsaw.

[4] Bex, F., Lawrence, J., & Reed, C. (2014, September). *Generalising argument dialogue with the Dialogue Game Execution Platform*. In COMMA (pp. 141-152).

[5] Bex, F., Snaith, M., Lawrence, J., & Reed, C. (2014). *Argublogging: An application for the argument web.* Web Semantics: Science, Services and Agents on the World Wide Web, 25, 9-15.

[6] Bex, F., Lawrence, J., Snaith, M., & Reed, C. (2013). *Implementing the argument web*. Communications of the ACM, 56(10), 66-73.

[7] Lawrence, J., Bex, F., Reed, C., & Snaith, M. (2012). *AIFdb: Infrastructure for the Argument Web.* In COMMA (pp. 515-516).

[8] Lawrence, J., Bex, F., & Reed, C. (2012). *Dialogues on the Argument Web: Mixed Initiative Argumentation with Arvina.* In COMMA (pp. 513-514).

[9] Rahwan, I., & Reed, C. (2009). *The argument interchange format.* In Argumentation in artificial intelligence (pp. 383-402). Springer, Boston, MA.

[10] Rahwan, I. (2008). *Mass argumentation and the semantic web.* Web Semantics: Science, Services and Agents on the World Wide Web, 6(1), 29-37.

[11] Rahwan, I., Zablith, F., & Reed, C. (2007). *Laying the foundations for a world wide argument web*. Artificial intelligence, 171(10-15), 897-921.

[12] Rowe, G., Macagno, F., Reed, C., & Walton, D. (2006). *Araucaria as a tool for diagramming arguments in teaching and studying philosophy*. Teaching Philosophy, 29(2), 111-124.

[13] Smith, S. P., & Harrison, M. D. (2002, April). *Improving hazard classification through the reuse of descriptive arguments*. In International Conference on Software Reuse (pp. 255-268). Springer, Berlin, Heidelberg.

[14] Kelly, T., & McDermid, J. (1998). *Safety case patterns-reusing successful arguments.*