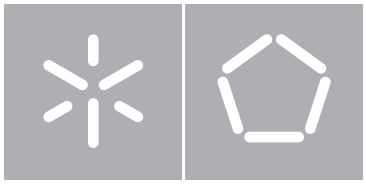
Universidade do Minho
Escola de Engenharia

Nuno Miguel Rocha de Sousa

Bidirectional Distributed Data Aggregation

Braga, October 31, 2014



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Nuno Miguel Rocha de Sousa

Bidirectional Distributed Data Aggregation

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho realizado sob a orientação de
Professor Manuel Alcino Cunha
Professor Paulo Sérgio Almeida

Braga, October 31, 2014

DECLARAÇÃO

Nome Nuno Miguel Rocha de Sousa

Endereço electrónico: rochadsouza@gmail.com Telefone: 910 564 967

Número do Bilhete de Identidade: 13199996

Título dissertação ☒ / tese ☐

Bidirectional Distributed Data Aggregation

Orientador(es):

Manuel Alcino Cunha e Paulo Sérgio Almeida

Ano de conclusão: 2014

Designação do Mestrado ou do Ramo de Conhecimento do Doutoramento:

Bidireccionalidade em cenários distribuídos

Universidade do Minho, 31 / 10 / 2014

Assinatura: Nuno Miguel Rocha de Sousa

ACKNOWLEDGEMENTS

I would like to express my gratitude to professor Alcino Cunha, my advisor in this master thesis, for his availability and interest demonstrated during the last year, for giving me all the necessary condition to complete my master degree, for his countless theoretical and technical contribution to this work, and for giving me the opportunity to join the HASLab.

To professor Paulo Sérgio Almeida, my co-supervisor, who likewise helped me in the realization of this work in sharing his knowledge and experience.

To Nuno Macedo, Tiago Jorge and Tiago Oliveira, my laboratory colleagues for the companionship, friendship, and help they have given me throughout the last years.

To my long time friends who always gave me their support, in good times and bad, who always encouraged me to press on and who always gave me amazing social moments needed to distract the mind from work.

Finally, a very special thanks to my mother, stepfather and sister for their patience and understanding during the writing of this thesis, for always support me and for always believe in me.

This work is funded by the ERDF through the programme COMPETE and by the Portuguese Government through FCT (Foundation for Science and Technology), project reference FCOMP-01-0124-FEDER-020532.

ABSTRACT

Transforming data between two different domains is a typical problem in software engineering. Ideally such transformations should be bidirectional, so that changes in either domain can be propagated to the other one. Many of the existing bidirectional transformation frameworks are instantiations of the so called lenses, proposed as a solution to the well-known view-update problem: if we construct a view that abstracts information in a source domain, how can changes in the view be propagated back to the source? The goal of a distributed data aggregation algorithm is precisely to compute in one or more network nodes a local view of a given global property of interest. As expected, such algorithms react to updates in the distributed input values, but so far no mechanisms were proposed to bidirectionalize them, so that updates in the computed view can be reflected back to the inputs. The goal of this thesis is precisely to research the viability of such bidirectionalizon in a distributed setting.

RESUMO

Transformação de dados entre dois domínios distintos é um problema típico em engenharia de software. Idealmente tais transformações deveriam ser bidireccionais, e assim essas modificações poderiam ser propagadas de qualquer domínio para o outro. Muitas das ferramentas existentes para transformação bidireccional são variações das chamadas *lentes*, propostas como solução para o bem conhecido problema *view-update*: se construímos uma vista que abstrai a informação de uma fonte de informação, como podem as modificações na vista serem propagadas de volta para a fonte de informação? O objectivo de um algoritmo distribuído de agregação é precisamente obter em um ou mais nodos da rede uma vista local resultante de uma dada propriedade global de interesse. Como esperado, tais algoritmos reagem a modificações nos valores de entrada, mas até agora nenhum mecanismo foi proposto para a sua bidireccionalização, ou seja, em para permitir modificações na vista obtida possam ser refletidas nos valores de entrada nos nós da rede. O objectivo desta tese é precisamente investigar qual a viabilidade dessa bidireccionalização em cenários distribuídos.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Part I

INTRODUCTORY MATERIAL

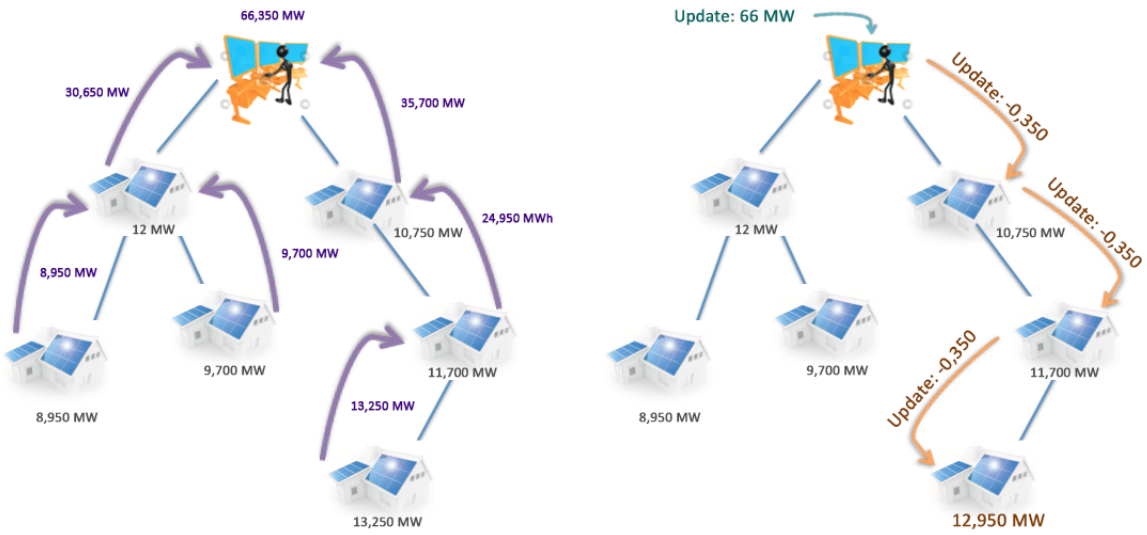
1

INTRODUCTION

Transformation and consistence maintenance between two different domains is a key issue in software engineering. For example, in Model Driven Engineering, relational database schemas are often derived from UML class diagrams, and it would be interesting that, when the user changes a schema, such changes are reflected back in the corresponding class diagram. Another example is the view-update problem in databases, where SQL queries can be defined to compute views of a database; it would be useful to be able to update such views and propagate back modifications to the database. One last example is when a platform-specific model (PSM) is generated from a platform-independent model (PIM); Several times the PSM must be changed due to new requirements, and the only option is to change the PIM according to the specification and generate (again) the PSM; It would be preferable to be able to change the PSM and have this update propagated to the PIM. All the above scenarios illustrate the need for bidirectional transformations (BXs), that allow, not only to derive a target artifact from a given source, but also to propagate changes in the target back to the source. Such bidirectionally is essential to keep both artifacts consistent, and avoid the need to manually propagate updates, a task that is both tedious and error prone.

This bidirectionality can be achieved in a naive way: writing two different transformations, which fit together in some appropriate sense, e.g., being somehow the inverse of each other. This option can be easily achieved but its maintenance is a difficult task because the two functions must embody the structure that the input and output schemas have in common, so changes to the schema will require a coordinated change in both. This approach is unsatisfactory for all but the simplest transformations. A more effective alternative is creating a notation in which both transformations can be specified in a single artifact, as advocated, for example, in the popular lens Foster et al. (2007) approach to bidirectional transformation.

Distributed data aggregation Jesus et al. (2011) - the capability to summarize information sublinear in the size of summarized data - is an important task in a distributed system, allowing the efficient collection of general information about the system. Some common aggregation functions are: *count*, *sum* and *average*, and some examples of application of this functions are: counting the number of nodes in the system; *sum* the storage available in the system; or system load average.



(a) Sum aggregation of energy production.

(b) Update sum value.

Figure 1.: Smart grid application scenario

This work aims to achieve a bidirectional and decentralized control in a distributed system, by combining the lenses approach with distributed data aggregation algorithms. This will allow, after aggregating the values from the different nodes, to make an update to the aggregate and have this change propagated back to one or more source nodes. To achieve this, a language will be proposed to describe and perform such bidirectional aggregation. Figure 1 presents an example of a possible application of such bidirectional data aggregation. Suppose we have a Smart Grid with a set of nodes, and we want to compute (in a given control node) the sum of energy production. This can be achieved with a simple distributed data aggregation algorithm, as depicted in Figure 1a. Our goal is to allow the control node to update this value and somehow propagate back the update to the source nodes, such that running the aggregation again yields the desired value, as shown in Figure 1b. Of course there are many possible ways to perform such *disaggregation*: to deal with this issue we intend to use the *least change principle* to control such non-determinism: this principle requires ways to select the ”best” among the possible options, to keep the selection as predictable as possible and therefore more deterministic.

The proposed approach can be applied in real uses cases like Smart Grids for intelligent use of energy or to the management of a Distributed Database. In Smart Grids, the main goals of such system are to manage the supply and energy production of network elements (houses, factories, and so on). More concrete examples are managing the energy produced (by photovoltaic panels in the network elements) and managing the energy consumption at a given time (hours where there is a greater energy consumption in the network). In Distributed Databases the goals of such system could be managing

the number of replicas, storage available and number of nodes in the system. With the proposed approach all these examples could be more easily implemented.

1.1 ORGANIZATION OF THE THESIS

The remainder of the thesis is as follows. In the next two chapters we provide background information about *Bidirectional Transformations* and *Distributed Aggregation*. In Chapter 4 we specify the application scenario, and present the system model and a compositional theory of *Bidirectional Aggregation*. Chapter 5 describes the created bidirectional distributed algorithm and Chapter 6 presents the developed simulator, which provides experimental results from the practical implementation of our theory about *Bidirectional Aggregation*. Finally in Chapter 7 we present our conclusions with potential paths for future research.

2

BIDIRECTIONAL TRANSFORMATIONS

The burgeoning interest in bidirectional transformations (BX's), concerning its enormous importance and applicability in a vast domains, has led to numerous proposed approaches due to different visions of the problems and class of problems where the need for BX arises. This section presents the main approaches of the existing BX frameworks to understand precisely their advantages and limitations.

2.1 FRAMEWORKS

The bidirectional transformations can be classified in three well-known and well-established frameworks: *Maps*, *Lenses* and *Maintainers*. These frameworks, explained below, may be classified according to its interface: *Symmetric*, "The update propagation nature is the same in both directions." (Pacheco et al. (2013)), i.e., the definition of forward and backward transformations are similar which means the two different domains are more balanced and contain roughly the same information, or that each may contain information not presented in the other; *Asymmetric*, "The update propagation nature is different in both directions." (Pacheco et al. (2013)), i.e., one of the domains contains more information than the other (one domain is an abstraction/ summary of the other). The following presentation assumes the BX is a pair of functions that implements a consistency relation between two different domains (Source (A) and Target (B)), whose main purpose is to propagate Source updates into Target updates, and vice-versa, to maintain the consistency between the two domains.

2.1.1 *Maps*

Maps are the simplest framework since they perform the translation of source updates into target updates and vice-versa without additional information. There is a forward transformation (1) named to that translates Source updates into Target updates, and the backward transformation (2) named from that translates Target updates into Source updates. The types of the Map transformations are:

$$\mathsf{to}\colon A \to B \tag{1}$$

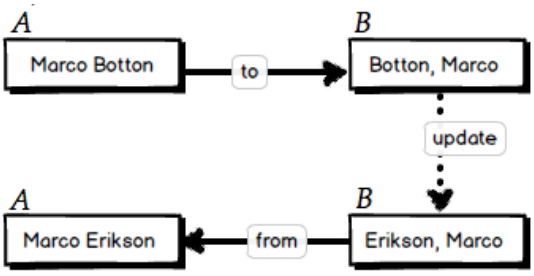


Figure 2.: Map example

$$\mathsf{from} : B \to A \tag{2}$$

The following laws are usually required for Maps to be well-behaved, but also entail that Maps are only useful for bidirectional transformation between bijective domains.

$$\mathsf{from}(\mathsf{to}(a)) = a \tag{3}$$

$$\mathsf{to}(\mathsf{from}(b)) = b \tag{4}$$

An example of the *Map* framework is presented in Figure 2 where two domains (A and B) store the names and surnames of persons: the domain A with the arrangement "Name Surname" and the domain B with the arrangement "Surname, Name". The consistency relationship between these two domains is: the name and the surname are the same in the both domains but in domain A the arrangement is "Name Surname" and in the domain B the arrangement is "Surname, Name". When an update is made (by user) in domain A, the from transformation will recover the consistency between the two domains.

2.1.2 *Lenses*

Lenses are the popular asymmetric framework proposed by Foster et al. (2007) inspired in the *view-update* problem from database theory. A *lens* l between A and B comprises two functions: a total $\mathsf{get}_l : A \to B$, that given an $a : A$ computes a view $b : B$; and a possibly partial $\mathsf{put}_l : B \to A \to A$, that given an updated view $b' : B$ and an $a : A$ *puts back* b' into a, in order to obtain an updated $a' : A$ of which b' is a view.

Lenses, as was mentioned, are inspired on the well-known view-update problem, so likewise view-update problem, the view (the result / output from get function) represents an abstract view from the

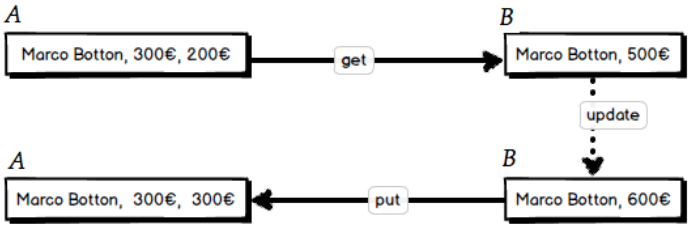


Figure 3.: Lenses framework: a non-deterministic example

Source with less information: this feature leads to non-determinism on the backward function as long as several elements from Source can be abstracted to the same view.

A non-deterministic example is depicted in Figure 3 where the domain A contains the following information: Name, monthly expenses for food and monthly expenses for rent (of certain person), and the domain B, which represents an abstract view from A with less information, contains the following information: Name and the sum of the monthly expenses. When an update is made on the sum of the monthly expenses value in domain B, there are a myriad of (correct) options to propagate this update to the domain A (e.g.: only update the expenses with food, only update the expenses with rent, update both equally, and so on). In this example (Figure 3), the used option was: only update the expenses with rent.

To deal with this non-determinism, the put function in addition to the updated element in Target considers additional *traceability* information, the original Source element, and the *least change principle* Meertens (2005) (detailed bellow).

In scenarios where an update is made in a view element and this new value has no counterparts in the original Source, there are two main solutions to taming with this issue: 1. Disallow the update. 2. The backward function invents a new Source element with a function `create` with the following type:

$$\mathsf{create} : A \rightarrow B \tag{5}$$

Due to the asymmetry of lenses, this framework only works well for surjective forward transformation, i.e., functions that only abstract away information.

Properties of lenses

A lens l between A and B is *well-behaved*, denoted by $l : A \trianglerighteq B$, if it satisfies the following laws:

$$\mathsf{put}_l\, b'\, a = a' \Rightarrow \mathsf{get}_l\, a' = b' \tag{PutGet}$$

$$\mathsf{get}_l\, a = b \Rightarrow \mathsf{put}_l\, b\, a = a \tag{GetPut}$$

Law PutGet ensures that the updates in b' are translated exactly, i.e. when put_l returns an $a' : A$ it is indeed a source value of which b' is a view (the lens is acceptable). Law GetPut ensures that if the view is not updated then put_l is bound to return exactly the same source that originated it (the lens is stable).

The totality in lenses means that the get function should be defined for any element in Source and the put function should be capable to translate and propagate to Source any update made in the Target. A lens l is total if both get_l and put_l are total.

As an example of a total well-behaved lens, consider $\mathsf{plus} : \mathbb{R} \times \mathbb{R} \trianglerighteq \mathbb{R}$, a lens that adds two numbers, with the following definition:

$$\mathsf{get}_{\mathsf{plus}}\,(x,y) = x + y$$

$$\mathsf{put}_{\mathsf{plus}}\, z\,(x,y) = \left(x + \frac{z-(x+y)}{2}, y + \frac{z-(x+y)}{2}\right)$$

In this case $\mathsf{put}_{\mathsf{plus}}$ divides the update on the view equally among both summands. However, this is just one among all possible well-behaved definitions for $\mathsf{put}_{\mathsf{plus}}$ – when get_l is not injective, if the view is updated, put_l may have several possible sources to choose from when translating an update. For example, the following less sensible definition for $\mathsf{put}_{\mathsf{plus}}$ is also well-behaved:

$$\mathsf{put}_{\mathsf{plus}}\, z\,(x,y) = \begin{cases} (x,y) \text{ if } z = x+y \\ (z,0) \text{ otherwise} \end{cases}$$

The problem is that PutGet, which establishes an upper bound on the behavior of put_l (i.e. which results are admissible), is for most applications too weak. One possibility to strengthen this law is to also require put_l to obey the *least change principle* Meertens (2005).

Least change principle in Lenses

The *least change principle* Meertens (2005), states that put should return a source a' that is the closest to the original source a, among all sources that share the same view b'. The specific metric used to evaluate *closeness* is of course type and application dependent, and should be left for the user to specify.

Assuming the source type A is a metric space (A, Δ), where $\Delta : A \times A \to \mathbb{R}$ is a distance function satisfying the standard properties

$$\Delta(a, a') = 0 \Leftrightarrow a = a' \tag{6}$$
$$\Delta(a, a') = \Delta(a', a) \tag{7}$$
$$\Delta(a, a'') \leq \Delta(a, a') + \Delta(a', a'') \tag{8}$$

the least change principle can be formalized as follows Macedo et al. (2013):

$$\mathsf{put}_l\, b'\, a = a' \wedge \mathsf{get}_l\, a'' = b' \Rightarrow \Delta(a, a') \leq \Delta(a, a'') \qquad \text{(LeastChange)}$$

When a lens is total, (LeastChange) subsumes (GetPut) since the source value a is the closest to itself, as property (6) requires (this property is usually known as *identity of indiscernibles*). A well-behaved lens $l : A \trianglerighteq B$ that also satisfies (LeastChange) for metric space (A, Δ) will be denoted as $l_\Delta : A \trianglerighteq B$.

Some examples of metric spaces on real numbers are

$$\Delta^{\ominus}(x_1, x_2) = |x_1 - x_2|$$
$$\Delta^{\odot}(x_1, x_2) = \mathsf{sgn}\, |x_1 - x_2|$$

where sgn is the standard signum function

$$\mathsf{sgn}\, x = \begin{cases} -1 & \text{if } x < 0 \\ 0 & \text{if } x = 0 \\ 1 & \text{if } x > 0 \end{cases}$$

Note that $\Delta^{\odot}$ is the *discrete metric* over $\mathbb{R}$, that returns 1 if the two values are different and 0 otherwise. A standard way to lift a metric space (A, Δ) to work on pairs (or any vector space) is to take the p-norm of the point-wise distance between coordinates. For example, the metric space $(A \times A, \Delta_p)$ thus obtained is defined as

$$\Delta_p((x_1, y_1), (x_2, y_2)) = \sqrt[p]{\Delta(x_1, x_2)^p + \Delta(y_1, y_2)^p}$$

Note that

$$\Delta_\infty((x_1, y_1), (x_2, y_2)) = \mathsf{max}(\Delta(x_1, x_2), \Delta(y_1, y_2))$$

Back to our example lens plus, for example, if the desired distance metric is $\Delta_2^{\ominus}$ then the first definition of $\mathsf{put}_{\mathsf{plus}}$ satisfies (LeastChange), but the second doesn't. However, if the desired metric is $\Delta_1^{\odot}$, for $\mathsf{put}_{\mathsf{plus}}$ to be well-behaved and satisfy (LeastChange) it would need to minimize the number of elements in the source pair that are changed. None of the above definitions satisfies that, but the following definition would:

$$\mathsf{put}_{\mathsf{plus}}\, z\, (x, y) = (z - y, y)$$

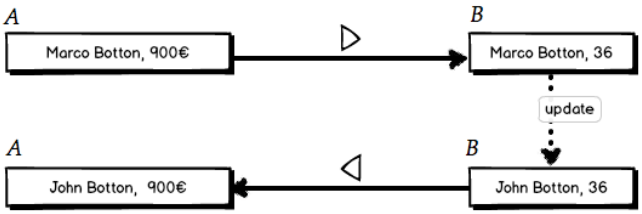


Figure 4.: Maintainers framework

Law PutPut

There is a (optional) third law that may be considered, called *PutPut*. This law states that the effect of two consecutive put applications is just the effect of the second put, i.e., the translation of a composite view does not depends on the intermediate source (is history ignorant).

$$\mathsf{put}(b', \mathsf{put}(b, a)) = \mathsf{put}(b', a) \tag{9}$$

A well-behaved lens that also satisfies this law, is called *very well-behaved*.

2.1.3 *Constraint maintainers*

Constraint maintainers are a symmetric framework that is represented by a pair of two functions that processes the necessary updates for constraint maintenance (Meertens (2005)), and its arguments are the original value at one end and the updated value at the other end. In opposition to Maps and Lenses, the consistence relationship is not implicitly in transformations, so the two functions know which parts are related between Source and Target and restore them after an update, up to a consistence relationship explicitly defined.

The types of the consistency relation R (10) between a source A and a target B and its maintainers (11 and 12) are represented below.

$$R \subseteq A \times B \tag{10}$$

$$\triangleleft : A \times B \to A \tag{11}$$

$$\triangleright : A \times B \to B \tag{12}$$

In figure Figure 4, the consistence relation states that in both domains the name must be the same. When the name is changed in one domain (domain B), the constraint maintainers function takes the original value at one end (Marco Botton) and the updated value at the other end (John Botton) and change the name on the other domain (domain A) to restore the consistence relationship between them.

Properties of constraint maintainers

The main requirement on constraint maintainers is that after an update at one end, the consistence can be restored after the assignment done from $\triangleright$ or $\triangleleft$ (depending on what end the update is made). This property is named as Correctness, and can be formalized as follows.

$$(x \triangleleft y, y) \in R \tag{13}$$

$$(x, x \triangleright y) \in R \tag{14}$$

Constraints maintainers must also ensure the minimal changes after an update (later, a more specific notion of minimal changes is presented). In a general way, a weaker version of minimal changes is that no unnecessary changes are made, i.e., if after an update to B if the consistence relation R remains valid, the effect of $x \triangleleft y$ should be nil. This property is named as Hippocraticness, and can be formalized as follows.

$$(x, y) \in R \Rightarrow x \triangleleft y = x \tag{15}$$

$$(x, y) \in R \Rightarrow x \triangleright y = y \tag{16}$$

The principle of Least Change

The principle of Least Change, as previously presented in *Lenses* section 2.1.2, requires that the transformation $\triangleright$ or $\triangleleft$, which establish the consistency between the two different domains (Source and Target), return minimal changes, up to some measure, when performing an update propagation. This principle requires ways to select the ”best” among the possible options, to keep the selection as predictable as possible and therefore more deterministic.

2.2 DEPLOYMENT OF BIDIRECTIONAL TRANSFORMATIONS

The bidirectionality can be achived with the following distinct techniques:

2.2.1 *Ad hoc*

The Ad-hoc approach consists in writing two functions (forward and backward), independently from each other, which must be proved correct by the developer: if there is a modification of the data types which the functions receive as input, its necessary to redefine both transformations - which is error-prone, and a new correctness prof is needed.

2.2.2 *Combinatorial*

In contrast to the Ad-hoc approach, the language used to express bidirectional transformations in a combinatorial framework has a native bidirectional semantics, i.e., each available primitive and combinator is bidirectional (has a forward and backward execution semantics). With this approach it is guaranteed that each expression is correct-by-construction. Lenses are the most well known example of a combinatorial bidirectional framework. An example of a well-behaved bidirectional lens primitive is the projection function $\mathsf{fst} : A \times B \to A$, defined as follows:

$$\mathsf{get}_{fst} : A \times B \to A \tag{17}$$

$$\mathsf{get}_{fst}(x, y) = x \tag{18}$$

$$\mathsf{put}_{fst} : A \to (A \times B) \to (A \times B) \tag{19}$$

$$\mathsf{put}_{fst}\, x'(x, y) = (x', y) \tag{20}$$

The fundamental lens combinator is sequential composition, that given two well-behaved lenses $l : A \trianglerighteq B$ e $m : B \trianglerighteq C$ produces a well-behaved lens $(l; m) : A \trianglerighteq C$, defined as follows:

$$\mathsf{get}_{(l;m)}\, a = \mathsf{get}_m(\mathsf{get}_l\, a) \tag{21}$$

$$\mathsf{put}_{(l;m)}\, c' a = \mathsf{put}_l(\mathsf{put}_m\, c'(\mathsf{get}_l\, a))a \tag{22}$$

2.2.3 *Syntactic*

In this approach, the backwards transformation is computed in compile-time, i.e., taking into account the syntax of a forward transformation, usually expressed in a general purpose programming language.

This approach is applicable when ”a deeper whole-program analysis is necessary or for the bidirectionalization of a (syntactically restricted) general-purpose language according to program transformation techniques” (Pacheco et al. (2013)).

2.2.4 *Semantic*

In this approach, the backwards transformation is computed in run-time, i.e., the decisions to translate and propagate the updates are made on-the-fly - ”programs are bidirectionalized on-the-fly, by encoding the unidirectional transformations as algorithms that observe the inputs/outputs for specific run-time executions.” (Pacheco et al. (2013)).

3

DISTRIBUTED DATA AGGREGATION

Data aggregation, "the ability to summarize information" quoting Robbert Van Renesse (Van Renesse (2003)), is an essential task in distributed systems that allows the determination of meaningful global properties in a decentralized manner. For instance, this task is the essential basis for many large networking services (e.g., address aggregation allows Internet routing to scale), the standard service in database (e.g., using SQL queries, users can explicitly aggregate data in one or more tables) and is used by many distributed paradigms and consistence mechanisms (e.g., synchronization based on voting requires votes counting).

The present section provides a more specific definition of Distributed data aggregation - considering that the process consists in the computation of an *Aggregation function* (Section 3.1), and provides a taxonomy of the existing distributed aggregation algorithms (Section 3.2).

3.1 AGGREGATION FUNCTION

As *aggregation function* takes a multiset $\mathbb{N}^A$ of values of type A and computes an output of type B. Since it aims at summarizing information, a value of type B should take less space then the input multiset, and it is often the case it takes much less space (for example, B being the same as A). In distributed data aggregation the input multiset abstracts a particular state of the network, where each node contains a value of type A. As such, it is convenient to refine the input multiset as a function from (unique) node identifiers to values of type A. The set of node identifiers of a network will be denoted as I, and thus the input to an aggregation function will have the type A^{I}. Node identifiers will be denoted by $i, j, i', \ldots$, and network states (i.e. multisets) by $s, s', \ldots$. To simplify the presentation we will usually denote the value of s at node i by s_i instead of $s(i)$.

In this thesis we will only consider aggregation functions where the input is a multiset of reals ($A = \mathbb{R}$). Typical aggregations that fall in this category include $\mathsf{Sum} : \mathbb{R}^{\mathsf{I}} \to \mathbb{R}$, $\mathsf{Min} : \mathbb{R}^{\mathsf{I}} \to \mathbb{R}$, $\mathsf{Max} : \mathbb{R}^{\mathsf{I}} \to \mathbb{R}$, $\mathsf{Average} : \mathbb{R}^{\mathsf{I}} \to \mathbb{R}$, $\mathsf{Mode} : \mathbb{R}^{\mathsf{I}} \to \mathbb{R}$, or $\mathsf{Count} : \mathbb{R}^{\mathsf{I}} \to \mathbb{N}$.

We can define a generic higher-order function that, likewise to *foldr* on lists, embodies a typical pattern of defining aggregations recursively over multisets:

$$(\!|\cdot,\cdot|\!) : (A \to B) \to (B \times B \to B) \to A^{\mathsf{I}} \to B$$
$$(\!|f, \oplus|\!)\ \{a\} = f\ a$$
$$(\!|f, \oplus|\!)\ (s_1 \uplus s_2) = \oplus((\!|f, \oplus|\!)\ s_1, (\!|f, \oplus|\!)\ s_2)$$

The first parameter f is used to compute the output value from the input at each node (notice that it cannot take the node identity into account), and $\oplus$ is a merge operator that combines the results of aggregating the values recursively at two disjoint partitions of the input multiset: note that $\uplus$ denotes the standard multiset sum, and since each input multiset can be decomposed in many different ways, for this function to be well-defined the operator $\oplus$ must be commutative and associative.

3.1.1 *Decomposability*

An aggregation function can be performed in a single computation (centralized), but in some cases it may need to be performed in a distributed way (e.g., due to efficiency issues). To perform a distributed aggregation, it is relevant to know the degree of decomposability in function, i.e., whether the function can be decomposed in several computations, with each computation receiving a sub-multiset as input. In order to clarify the notion of *decomposable aggregation function*, and its sub-set called *self-decomposable aggregation function*, the follow paragraph provides a more precise definition.

Decomposable and Self-decomposable

The main difference between decomposable functions and its subset self-decomposable functions is: in self-decomposable functions, the intermediate results can be calculated in the output domain, in opposition, decomposable functions, that are not self-decomposable, need an intermediate domain to hold the intermediate results.

An aggregation function is *self-decomposable* Jesus et al. (2011) if it can be defined with the above folding pattern. For example, Sum, Min, Max, and Count are self-decomposable:

$$\mathsf{Sum} = (\!|\mathsf{id}, \mathsf{plus}|\!)$$
$$\mathsf{Min} = (\!|\mathsf{id}, \mathsf{min}|\!)$$
$$\mathsf{Max} = (\!|\mathsf{id}, \mathsf{max}|\!)$$
$$\mathsf{Count} = (\!|\underline{1}, \mathsf{plus}|\!)$$

Here, id is the identity function and $\underline{k}\ a = k$ is a function that always returns a given constant.

The computation of two self-decomposable aggregations can always be fused in a single self-decomposable aggregation, since

$$\begin{aligned}(\!|f, \oplus|\!) \vartriangle (\!|g, \otimes|\!) = \quad & (\!|f \vartriangle g, \odot|\!) \\ & \textbf{where } \odot((x_1, y_1), (x_2, y_2)) = (x_1 \oplus x_2, y_1 \otimes y_2))\end{aligned}$$

Here, $(g \vartriangle h)\ x = (g\ x, h\ x)$ is the split combinator that packs the output of two functions in a pair. This law is the equivalent for multisets of the well known *loop fusion* or *banana split* law for folds over lists Meijer et al. (1991).

An aggregation function g is *decomposable* Jesus et al. (2011) if it can be defined as the composition of a function h after a self-decomposable aggregation, i.e. $g = h \circ (\!|i, \oplus|\!)$. For example, Average is a decomposable aggregation, since:

$$\begin{aligned}\mathsf{Average} = \quad & \mathsf{div} \circ (\mathsf{Sum} \vartriangle \mathsf{Count}) \\ & \textbf{where } \mathsf{div}(s, n) = s/n\end{aligned}$$

Some aggregations are not decomposable, e.g. Mode.

A practical example is: given a multiset $N = \{1, 2, 3, 4, 5\}$ and the submultisets $N' = \{1, 2, 3\}$ and $N'' = \{4, 5\}$ from N, a self-decomposable function (Sum) may calculate the result directly in the output, as shown in Figure 5a. A decomposable function (Average), which is not self-decomposable, may not calculate the result in the output domain, as in Figure 5b, i.e., it needs an intermediate domain with more information, as illustrated in Figure 5c - in this case, the number of elements that the average value corresponds.

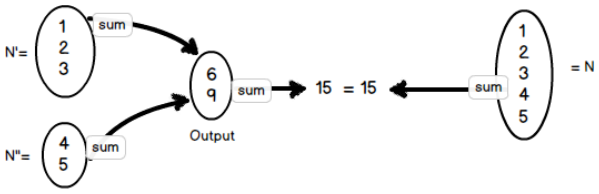
3.1.2 *Duplicate sensitiveness and idempotence*

For some aggregation functions it may be relevant whether a given value occurs several times in a multiset. For aggregation functions such as min and max the occurrence of a given value several times is not a problem because it does not influence the final result, which only depends on its support set (the set obtained when all the duplicate values are removed from the original multiset), as exemplified in Equation (23) for Max.

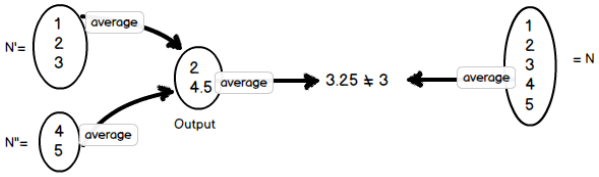
$$\mathsf{Max}(\{1, 1, 2, 2, 3, 3\}) = \mathsf{Max}(\{1, 2, 3\}) = 3 \tag{23}$$

$$12 = \mathsf{Sum}(\{1, 1, 2, 2, 3, 3\}) \neq \mathsf{Sum}(\{1, 2, 3\}) = 6 \tag{24}$$

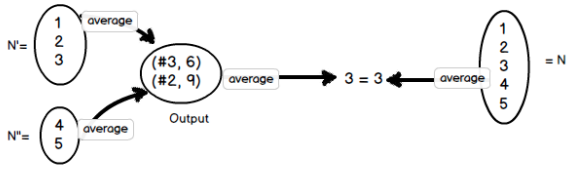
Duplicate sensitiveness is relevant in a distributed context because many duplicate insensitive functions can be implemented using an idempotent binary operator on the elements of the multiset, that



(a) Self-decomposable function



(b) Decomposable function which is not self-decomposable



(c) Decomposable function which is not self-decomposable

Figure 5.: Decomposable and Self-decomposable examples

helps obtaining fault tolerance and decentralized processing, allowing redundancy by retransmission or sending values across multiple paths without affecting the final result of the aggregation function.

3.2 TAXONOMY

In this section, we present a general taxonomy for existing distributed data aggregation algorithms. The following taxonomy will classify the algorithms according to two perspectives: *Communication* and *Computation* (Jesus et al. (2011)). The communication viewpoint refers to issues such as routing protocols and network topologies, and the computation viewpoint refers to the aggregation functions performed by the algorithms.

3.2.1 *Communication perspective*

This viewpoint, which categorizes algorithms according to its routing protocols and network topologies, may be divided in three main categories: *Structured, Unstructured* and *Hybrid*.

Structured

These algorithms are ”structure-dependent”, i.e., depend on a specific network topology and routing topology to perform correctly. The main cons of these algorithms are: the adaptability to dynamic scenarios and become directly affected by problems from the used routing topology (on tree-based communication structures, nodes are a point of failure). The first con, can be overcome with an additional preprocessing phase that ”creates” a routing topology before executing - but this solution may be ”heavy” to achieve.

HIERARCHY-BASED APPROACHES

The main feature of these algorithms is the routing strategy which is hierarchically organized (e.g.,tree), i.e., there is a node where all the operations in system are triggered from - called *sink*.

Commonly these algorithms spread the information level to level, up to an established hierarchy, and this process is made in two phases: *Request* and *Response*. *Request* refers to the aggregation request, which was triggered from sink, through all the network; *Response* refers to the answer from all nodes in the system to the request previously performed. The strategy used in this approach is simple and therefore provides an accurate execution of aggregates (if there isn't node failures during the aggregation process).

RING-BASED APPROACH

In general, in ring-based approaches, the data is propagated in a ”circular manner”, due to the topology. A failure in a ring can compromise the communication in all system (single point of failure). Typically this approach is surpassed by the hierarchy-based approach.

Unstructured

In opposition to structured algorithms, unstructured algorithms are not ”structure-dependent” and can operate independently from the network topology. This category, usually gossip-based, is important to solve problems in large scale systems such as node failure and message loss.

FLOODING / BROADCAST BASED APPROACHES

In this approach all the nodes in the system participate in the data aggregation process, i.e., the node which performs the request sends messages to all neighbors and, in turn, neighbors do the same. Typically this approach brings a high network load during the aggregation process.

RANDOM WALK BASED APPROACH

This approach only promotes the participation of some nodes and relies on probabilistic methods, i.e., only process data samplings to estimate an aggregation value. In general, the messages are sent sequentially from one node to another (to a neighbor) and this neighbor is randomly selected (called one-to-one). Due to the partial participation of the nodes in the system, this approach introduces a

lower network load during the aggregation process, nonetheless, the result will always contain an estimation error.

GOSSIP-BASED APPROACH

Gossip-based algorithms, in general is a midpoint of the two previous approaches, i.e., send a request to a subset that is randomly chosen. A start node sends to some neighbors the request message and, in turn, the neighbors do the same process (called one-to-many). The main features of this approach are: simplicity, robustness in terms of fault tolerance and scalable information propagation.

Hybrid

This category of algorithms mixes the two previous categories to achieve solutions that combines their pros and minimize their cons, in order to obtain better solutions.

HYBRID APPROACHES

In general, this hybrid approach is the combination of hierarchy based schemes (which are more affected by the occurrence of faults) and gossip based algorithms (which introduce a higher overhead during the aggregation process) in a solution that is a trade-off between this two approaches.

3.2.2 *Computation perspective*

This viewpoint, which categorizes algorithms according its aggregation functions, may be sub-divided in the next categories: Hierarchical, Averaging, Skecthes, Digests, and Sampling. We will now describe the main characteristics of these different categories.

Hierarchical

In hierarchical approaches, inputs are divided into separated groups and the calculation is performed distributively and hierarchically (hierarchy previously calculated), taking advantage of decomposable property in some aggregation functions. This approach performs correctly if no faults occur during the aggregation process (no fault tolerance) and the global processing and memory required are identical to the ones required from a centralized application that performs an aggregation function.

Averaging

Averaging approaches, in general, perform the calculation over partial aggregates, and continuously average and exchange data among all the nodes that participates in the aggregation process. This approach tends to achieve high accuracy (with all nodes converging to the correct result) but only if algorithms respect an important principle - *mass conservation* (the *sum* of the values of all nodes in the system is the same along time). Usually this method can be found in gossip based approaches.

Skecthes

The main feature in this approach is the use of an auxiliary structure (with a fixed size) that holds a sketch off all node values. Being the operations in sketches order and duplicate insensitive, the aggregation process can be performed through multiple paths and therefore independent from the routing topology. This approach is based on probabilistic methods and the accuracy depends on the sketch size, i.e., there is a trade-off between accuracy and sketch.

Digests

The main feature in this approach is the ability to perform complex aggregation functions (e.g., median) besides the common ones (Sum, Count). In general, these algorithms calculate a *digest* that abstracts the system and holds an approximation of the probabilistic distribution of the input values in all nodes. The accuracy depends on the quality of *digest* calculated.

Sampling

Sampling approach refers to a set of distributed algorithms that was designed uniquely for one aggregation function - Count. This kind of approach exists due to the importance of the Count function (e.g., determination the number of nodes in the system). These algorithms are based on probabilistic methods and only a portion of nodes in the network participates in the aggregation process, therefore, it is lightweight in terms of messages exchanged.

Part II

CONTRIBUTION

4

BIDIRECTIONAL DISTRIBUTED DATA AGGREGATION

In a distributed setting it could be interesting to bidirectionalize some aggregation algorithms to achieve a novel mechanism to control the overall state of the network. This would allow changes in the aggregate value to be propagated back to the source nodes, ensuring convergence to a new desirable state. In this chapter, we first present some potential application scenarios for such bidirectional distributed aggregations followed by the specifications of the used system model and the description of some *least-change* metrics which establish the different behaviors in the disagregation process. Finally, we motivate the need to control the causality of messages in the system, and propose an algorithm to perform bidirectional distributed data aggregation that takes into account that and other issues identified in the previous sections.

4.1 APPLICATION SCENARIOS

Suppose there is a Smart Grid where its nodes are energy production sources, more specifically hydroelectric dams. The determination of a meaningful property in the system at some moment, for instance the produced energy, is very important to take some decisions about the system, such as: ”Should we increase the production of energy?” or ”Should we decrease the production of energy in a particular damn?”. This can be achieved with a simple data aggregation algorithm. However, how can we update the system state after obtaining such information? A simple option is to calculate an optimal solution to the system in a centralized manner, and send the optimal solution to all nodes in the system. This solution is not practicable for large systems because the central processing of the optimal solution may be heavy and the amount of information (information with all the system state at some moment) that is passed on the network may overload the network. The approach proposed in this thesis, bidirectional distributed data aggregation, intends to enforce updates on the system state in a decentralized manner (the updates are calculated in each node with a local perspective) and with the minimal amount of information passing on the network.

Another example is on distributed databases where replication plays an important role - due to backup and high-availability issues. Suppose that in a certain distributed database are stored videos (e.g., Youtube), and due to the success of a particular video it would be convenient to increase the

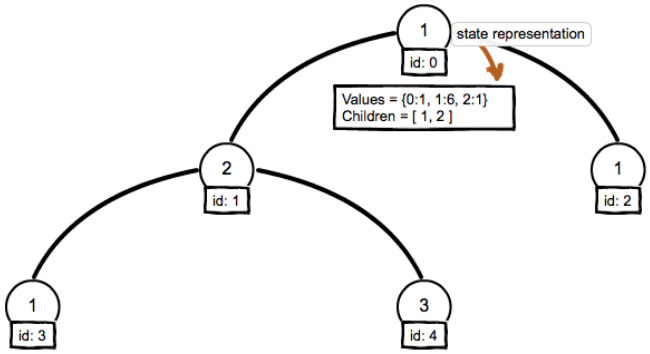
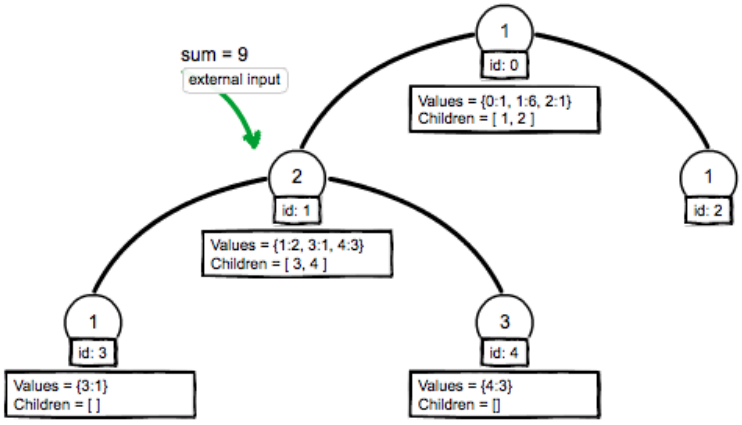


Figure 6.: Node state and a representation of the state about the down level nodes

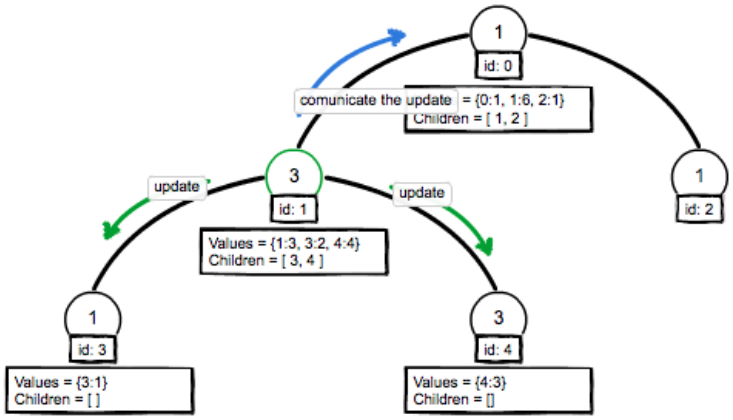
number of replicas of this video - to achieve a better high-availability and performance configuration. A classic solution to achieve this goal would be to run an aggregation algorithm for the determination of the number of replicas of that video, somehow determine which servers don't have a replica and send instructions for these servers to store a new replica of that video. With the approach proposed in this thesis, bidirectional distributed data aggregation, after computing the number of replicas of the video in the system (through aggregation), only needs to define the new desired number of replicas and the system would converge to the desired state - in a transparent manner.

4.2 SYSTEM MODEL

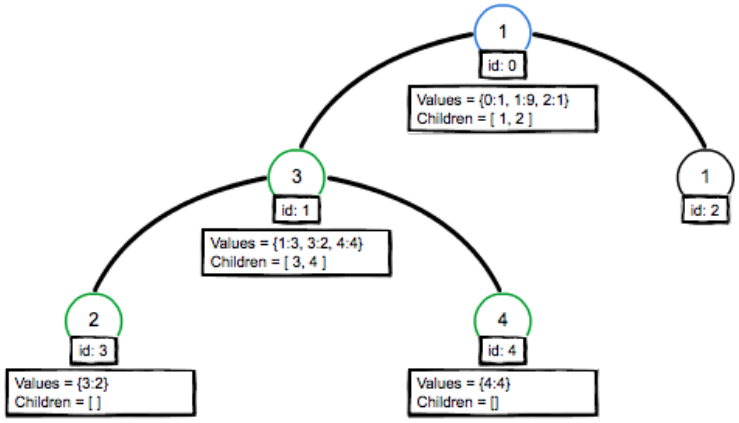
In this project it is assumed that the system model has the follow specification: tree topology, asynchronous communication, no faults occur and all nodes are reactive, in other words, they react to external inputs. Tree topology was chosen due to its hierarchy structure that simplifies the communication with all the nodes in the system - at the routing level, i.e., when it is necessary to communicate with some node in the system, there is only one path to reach such node. The asynchronous communication is assumed, and so the communication is not accomplished according to some clock - giving more flexibility to the system at the communication level. In a first stage we do not consider fault-tolerance and the system is static (system does not supports the addition and removal of nodes) so that the focus of this project is centered on the bidirectionalization of distributed aggregation primitives rather then system faults and system management issues. All the nodes in the system keep their state and a representation of the state about the down level nodes (Figure 6), and they react to external inputs (Figure 7a) by updating themselves and the down level nodes (Figure 7b), and also forwarding this new state to upper level nodes (till reaching the root node) (Figure 7c). From a more abstract standpoint, the system is a Map from identifiers I to values (usually $\mathbb{R}$) and thus resembling a multiset of values where each node is represented by a pair $(i, value)$.



(a) External input: ”Sum=9”



(b) Update node and the down level nodes



(c) Communication of the new state to upper level nodes

Figure 7.: Representation of an update in the system triggered by an external input

The aggregation primitives that are considered to be bidirectionalized are: Count, Sum, Max and Min, and the used aggregation algorithm is *Structured - Hierarchy-based* in a computational viewpoint and *Hierarchal* in a communication viewpoint due to the chosen topology - tree topology. The BX framework used are *Lenses* because, as was previously referred, each node represents an abstract view with less information from the down level nodes and itself, and the use of the *principle of least change* Meertens (2005) in this context may be applied in the following scenario: when there is an update in the system, the changes in the system must be minimal according to some metric, for instance, if the user wants to increase the energy production of the system (on the hydroelectric dams example) with the minimal number of changed nodes (due to cost issues), here the used metric to evaluate the *closeness* to the desired result is the number of changed nodes.

4.3 BIDIRECTIONAL AGGREGATIONS

Not taking into account the least-change principle, aggregations can be seen as lenses if their parameter functions are also lenses, as the following proposition shows.

Proposition 1. *If g and $\oplus$ are well-behaved lenses, then $(\!|g, \oplus|\!)$ is also a well-behaved lens.*

$$\frac{g : A \trianglerighteq B \quad \oplus : B \times B \trianglerighteq B}{(\!|g, \oplus|\!) : A^{\mathsf{I}} \trianglerighteq B}$$

The proof is simple assuming the following definitions for get *and* put.

$$\begin{aligned}
&\mathsf{get}_{(\!|g,\oplus|\!)}\{a\} = \mathsf{get}_g\, a \\
&\mathsf{get}_{(\!|g,\oplus|\!)}(s_1 \uplus s_2) = \mathsf{get}_{\oplus}(\mathsf{get}_{(\!|g,\oplus|\!)}\, s_1, \mathsf{get}_{(\!|g,\oplus|\!)}\, s_2)
\end{aligned}$$

$$\begin{aligned}
&\mathsf{put}_{(\!|g,\oplus|\!)}\, b\, \{a\} = \{\mathsf{put}_g\, b\, a\} \\
&\mathsf{put}_{(\!|g,\oplus|\!)}\, b\, (s_1 \uplus s_2) = \quad \mathsf{put}_{(\!|g,\oplus|\!)}\, b_1\, s_1 \uplus \mathsf{put}_{(\!|g,\oplus|\!)}\, b_2\, s_2 \\
&\qquad\qquad\qquad\qquad\quad \textbf{where}\ (b_1, b_2) = \mathsf{put}_{\oplus}\, b\, (\mathsf{get}_{(\!|g,\oplus|\!)}\, s_1, \mathsf{get}_{(\!|g,\oplus|\!)}\, s_2)
\end{aligned}$$

□

Given two network states A^{I} and metric space (A, Δ) we can compute a $|\mathsf{I}|$-vector with the pointwise distance between them. Likewise for pairs, (A, Δ) induces a standard family of metric spaces (A^I, Δ_p) by taking the *p-norm* of such vectors:

$$\Delta_p(s, s') = \sqrt[p]{\sum_{i \in \mathsf{I}} \Delta(s_i, {s'}_i)^p}$$

This proposition is not correct if we want the resulting *lens* to be a *least-change lens*, thus it is not possible obtain a generic and trivial disaggregation algorithm which merely performs the put (from the function which computes the new local view, i.e., $\mathtt{put}_{\oplus}$) in the intermediate nodes to distribute the new update among the children (and for itself) and the put (from the function which computes the view from the input change, i.e., $\mathtt{put}_{\mathtt{g}}$) to know which value to assign to the node.

Nonetheless, if the $\oplus$ takes into account extra information about the system, in particular, the size of the subnet that the node is root of, is it possible to implement the disaggregation of some primitives and some *least-change* criteria.

4.4 LEAST-CHANGE METRICS

An algorithm which provides bidirectionalization to a distributed system should be designed in a modular way, to support evolution and maintenance, e.g. addition of new features to the algorithm like primitives or metrics. In this section we present the chosen metrics to evaluate the *closeness* of the produced results in disagregation (and the limitation that some of them have in its implementation).

As mentioned previously, the *aggregation* ($\mathtt{get}_l$) may be not injective and thus the *disaggregation* ($\mathtt{put}_l$) can produces a myriad of correct results. In order to achieve a more predictive disagreggation, $\mathtt{put}_l$ will be required to obey the *least change principle*, and we will specify several metrics to evaluate the closeness of the produced results.

4.4.1 *Sum of squared differences*

In order to change all the elements in the system equally, this metric evaluates the sum of squared differences in the new system state (assuming $p = 2$).

$$\Delta^{ssd}(s, s') = |s' - s| \tag{25}$$

$$\Delta_p^{ssd}(s, s') = \sqrt[p]{\sum_{i \in \mathsf{I}} (|s'_i - s_i|)^p} \tag{26}$$

HYDROELECTRIC DAMS In the hydroelectric dams example, this metric can be applied when at some moment there is the need of increase the amount of energy to be produced by the network and the user wants this increase be distributed evenly among the dams.

DISTRIBUTED DATABASE SYSTEMS In a distributed database systems, this metric can be applied when there is a need to increase the number of replicas of n different records and the user wants this new replicas be distributed evenly across the servers.

4.4.2 *Sum of relative deviations*

This metric evaluates the changes in disagreggation concerning to the relative deviations.

$$\Delta^{srd}(s, s') = \frac{|s' - s|}{s} \tag{27}$$

$$\Delta_p^{srd}(s, s') = \sqrt[p]{\sum_{i \in \mathsf{I}} \left(\frac{|s'_i - s_i|}{s_i} \right)^p} \tag{28}$$

Although this metric does not obey the *symmetry* rule of metrics, and thus considered *quasi-metric*, this fact seems to have no impact in our goal.

Application scenario

HYDROELECTRIC DAMS In the hydroelectric dams example, this metric can be applied when at some moment there is the need to increase the amount of energy to be produced by the network and the user wants this increase to be equally distributed in a relative way. In scenarios where the amount of energy produced in each dam is related with the capacity of the dam to produce energy, this metric may be useful (e.g. if at a given time each dam in the system is producing 30% of its capacity and the user wants to increase the overall production and keep this relation).

DISTRIBUTED DATABASE SYSTEMS In database systems, when there is the need to increase the number of replicas of n different records and the user wants this new replicas be equally distributed, in a relative way, due to the storage capacity of each server (compared to the example given in the hydroelectric dams).

4.4.3 *Sum of Changed nodes to Zero or Non-Zero*

To "turn on or off" nodes in the system can have much higher cost then changing the values at the nodes, e.g. the cost in turn on or off dams, so this metric gives more weight to the changes from zero to non-zero though continues to weight the difference of values.

$$\Delta^{cnz}(s, s') = \frac{|s' - s|}{s'^2 + s^2} \tag{29}$$

$$\Delta_p^{cnz}(s, s') = \sqrt[p]{\sum_{i \in \mathsf{I}} \left(\frac{|s' - s|}{\sqrt{(s'_i)^2 + (s_i)^2}} \right)^p} \tag{30}$$

HYDROELECTRIC DAMS In the hydroelectric dams example, turn on a damn may mean an increase in the maintenance costs due to the startup of the engines of dams. This metric can be applied to prevent the updates in the network to turn off dams and thus avoid future extras costs in turning them back on again.

4.4.4 *Sum of Changed Nodes*

Sum of changed nodes evaluates the number of elements that were changed in disagreggation.

$$\Delta^{cn}(s, s') = \mathsf{sgn}\left(|s' - s|\right) \tag{31}$$

$$\Delta_p^{cn}(s, s') = \sqrt[p]{\sum_{i \in \mathsf{I}} \mathsf{sgn}\left(|s'_i - s_i|\right)^p} \tag{32}$$

HYDROELECTRIC DAMS In the hydroelectric dams example, update the amount of energy produced by certain dam may represent extra costs due to the energy used to do this increase. This metric performs the disaggregation, changing the least number of dams as possible.

DISTRIBUTED DATABASE SYSTEMS In a distributed database systems, the addition of new records in a certain server may represent a momentary unavailability of the services of that server, and thus, postpone the answers to the current requests. This metric is useful in this case because it performs the disagregation of adding the new records using the least number of servers as possible.

In these examples, as the metric is only to evaluate if the nodes where changed (or nor), we are only interested in the case $p = 1$.

4.4.5 *Examples in using different metrics with the same primitive on disagreggation*

Here, we present an application scenario, where, after the aggregation with a given primitive, the disaggregation is performed with different metrics, allowing us to compare the impact of the metric in the final result.

The example illustrates a hydroelectric dam network where after an aggregation with Sum (Figure 8), to compute the amount of produced energy (Figure 9)), this amount is updated and disaggregated with the *Sum of Changed of squared differences* (Figure 9a) and *Sum of relative deviations* (Figure 9b) metrics.

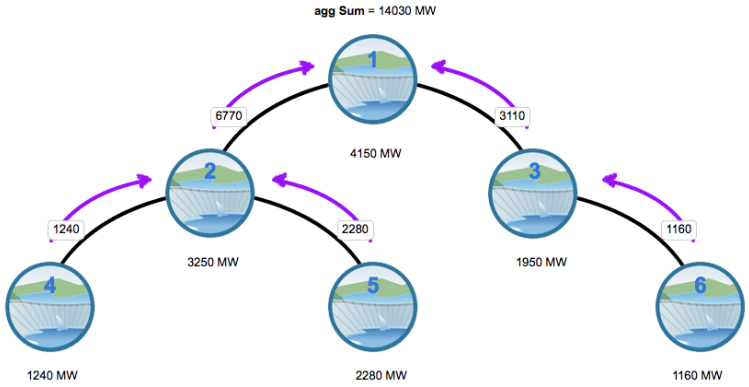
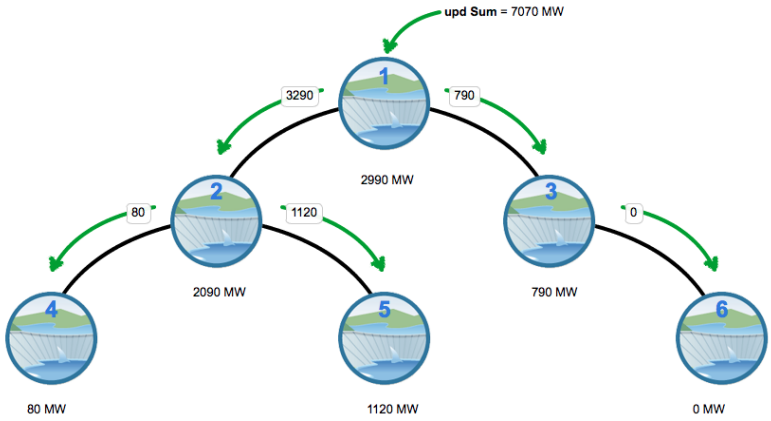
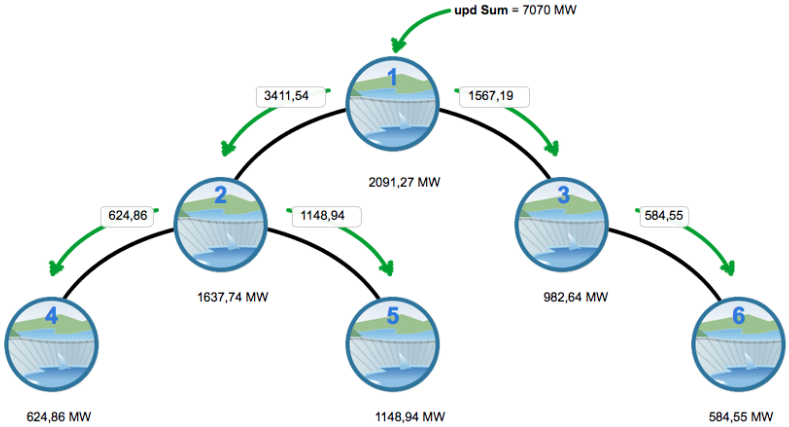


Figure 8.: Representation of disagreggation of the same primitive with different metrics (aggregation)

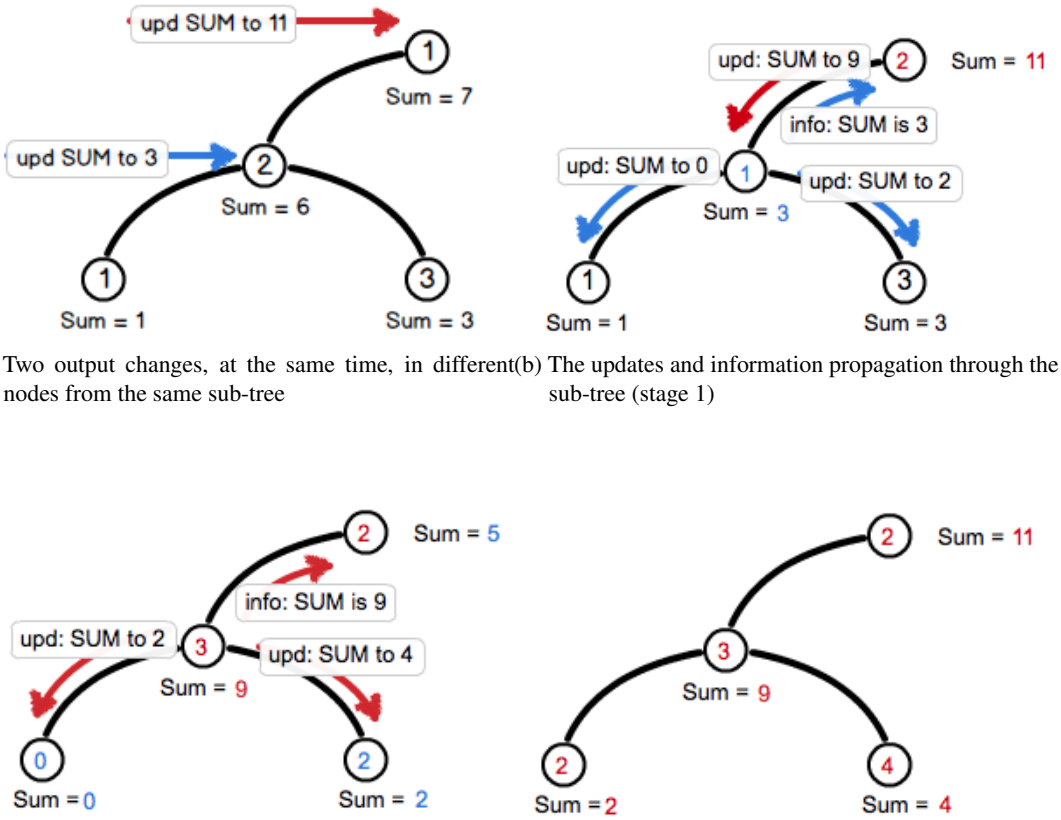


(a) Disagregation of Sum with *Sum of squared differences* metric



(b) Disagregation of Sum with *Sum of relative deviations* metric

Figure 9.: Representation of disagreggation of the same primitive with different metrics



(a) Two output changes, at the same time, in different nodes from the same sub-tree

(b) The updates and information propagation through the sub-tree (stage 1)

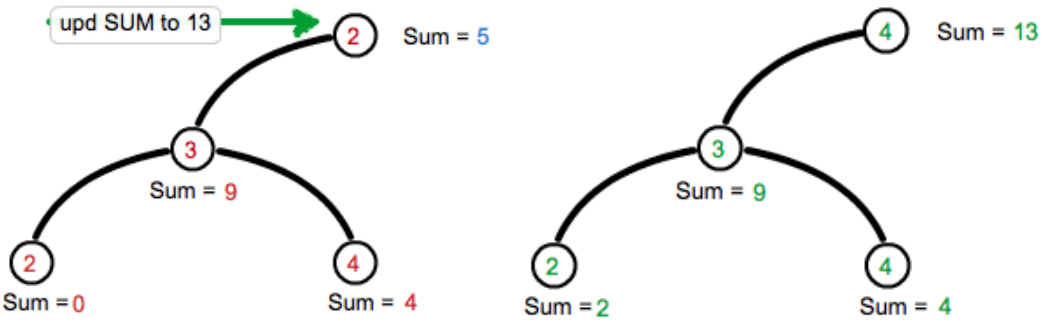
(c) The updates and information propagation through the sub-tree (stage 2) — Here, with no version control, there is a glitch in the root node *(sum = 5)*

(d) The final (correct) state

Figure 10.: Output Change - cross information

The requests in the system can be made in any node, and when an *Output Change* is requested in some node and at the same time there is another *Output Change* in another node (with the same primitive), the request made in the topmost node, i.e. the node which contains the other node in its sub-tree, should win, and so the system state reflect the changes performed by request in the topmost node.

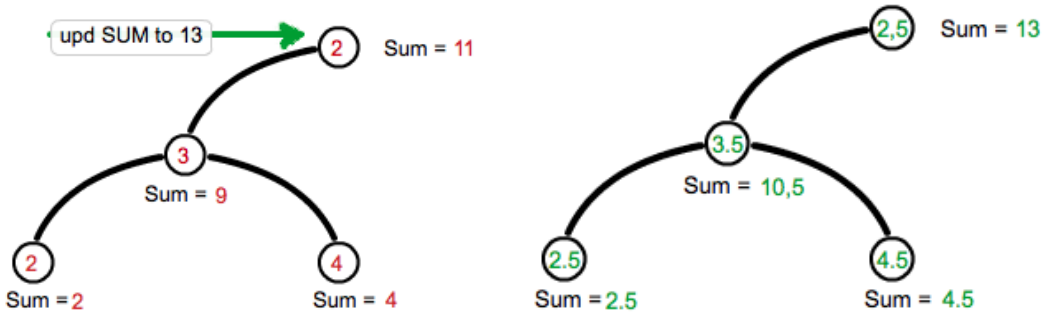
However, in a naive distributed disaggregation algorithm, which does not consider the causality of the exchanged messages, when there is an *Output Change* in some node and if before this operation is finished there is another *Output Change* in another node bellow the node of the first request, it is verified a *glitch* which is a transient fault in the system that eventually corrects itself (i.e. eventually the system converge to a correct state, figure 10). As can be seen in Figure 10c the Sum in root node is incorrect which is eventually corrected in the future due to the exchanged messages in the system



(a) Output change requested to the root node (with *glitch* Sum = 5)

(b) Final state of the system after process the request

Figure 11.: Output Change with *glitch*



(a) Output change requested to the root node (without *glitch*))

(b) Final state of the system after process the request

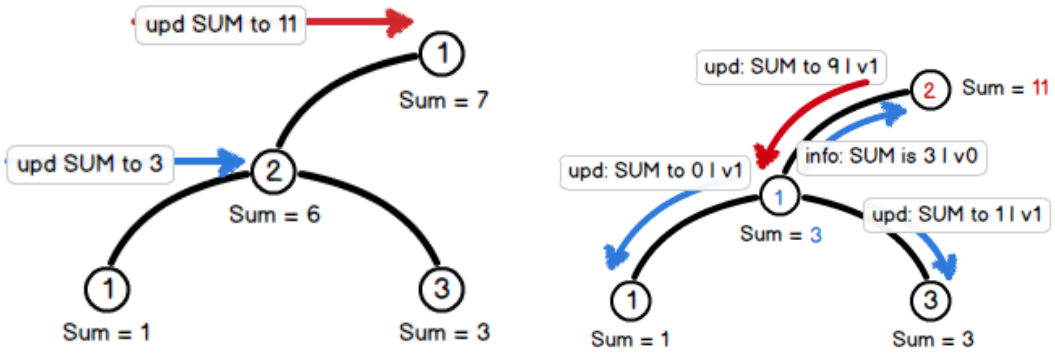
Figure 12.: Output Change without *glitch*

(Figure 10d).

During the time of the *glitch*, if there is an external request in the node where is the glitch (in our example of Figure 10c, in the root node), the final state of the system will be correct but the distance (amount of changes in the system concerning to some metric) from the current state to the end state is greater or equal than if there is no glitch. Using the example in Figure 10, in Figure 11 an *Output Change* is requested in the root node where there is a *glitch* and in the Figure 12 the same *Output Change* is requested in the root node but without a *glitch*. As can be seen both final states are correct but different and the distance from the initial state to the end state in Figure 11 is greater than the example in Figure 12 (in this example we assume the *Sum of squared differences* metric), as the following calculation shows.

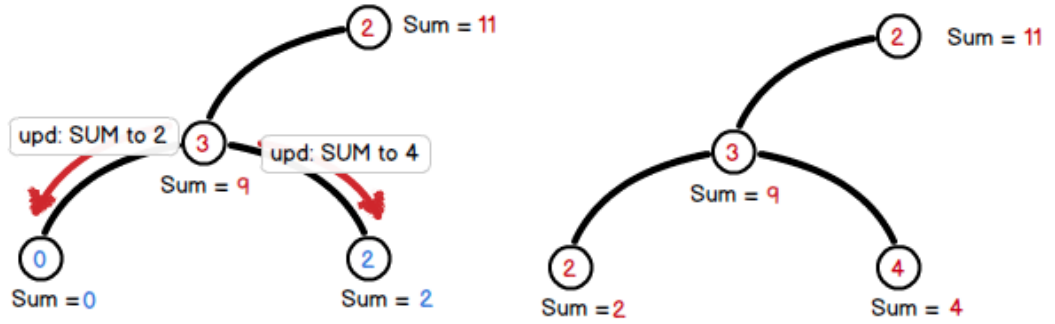
$$
\begin{aligned}
dist_{glitch} &= (4-2)^2 + (3-3)^2 + (2-2)^2 + (4-4)^2 \\
&= 4 \\
dist_{noglitch} &= (2.5-2)^2 + (3.5-3)^2 + (2.5-2)^2 + (4.5-4)^2 \\
&= 1
\end{aligned}
$$

4.5.1 *Counter system*



(a) Two output changes, at the same time, in different nodes from the same sub-tree

(b) The updates and information propagation through the sub-tree (stage 1)



(c) The updates and information propagation through the sub-tree (stage 2).

(d) The final (correct) state

Figure 13.: Output Change - cross information

To tame the causality of the messages a *Counter system* was implemented to assign to messages a value, called version, and thus when another node receives a message it knows its causality/ order and can take the correct decision concerning it: process the message or discard it (Figure 13).

Thus, each node has a counter to generate the version number for its messages and saves the version number of the parent (if any) and when it sends a message to its parent, this message contains the version number of the last message received from the parent, and when it sends a message to its child(ren) this counter is increased and sent with the message. Hence, when the parent receives the message, it compares the message version number with its counter value and if they are equal, processes the message (i.e., the child saw the last message of the parent), otherwise, discard it. When a child receives a message from its parent, it processes the message and updates the version number of the parent (i.e., the received message is always newer than the last message received from the parent because it is assumed no message loss in the system).

4.6 ALGORITHM

The developed algorithms have as main feature a compositional approach leading to better understanding and organization. The main functions are detailed and described bellow and the auxiliary functions have a suitable name for the better understanding of their role.

The first algorithm is for *Sum of Changed Nodes*, *Sum of Changed nodes to Zero or Non-Zero* and *Sum of squared differences* metrics because in this metrics the *extra values*, used to calculate the new view for the node, is known and static (the Count property, because as was assumed, the system does not supports the addition and removal of nodes), and thus, do not need to wait for the system to converge to the new state to calculate the new *extra values* - which is verified in the second algorithm. In this algorithm we assume that when the algorithm initializes, the Count property is known and is available from the beginning, the $aggregation$ occurs when there is *input changes* in nodes and, the *output change* only can be executed at some node after this node has a view of the sub-system bellow it - the aggregated values.

As constants, this algorithm has in each node:

- p_i which represents the parent of the node (if the node has parent, e.g., the root node has no parent);
- N_i is the set that contains the children's id of the node;
- E_i is the extra values to support the calculation of the new local views in the disagregation process. As already mentioned, this value is known from the beginning due to the system being static;

As state in each node we have:

- a_i, contains the value stored in the node;

- V_i, is a vector which contains the view of the node (of its own stored value) and the local views of its children.

- C_i, is a vector which contains the counter value of the node and of its parent (if it has a parent).

The lenses that are parameters of the algorithm, and are instantiated in the table 1, are:

- The *lens* $\mathtt{f}$, where the get function computes the view from the value stored in the node (e.g., if the value stored in the node is two geographical points and we want the view to be the distance between them) and the put function takes a view and the current stored value and computes a new value, to be stored in the node;

- The $\oplus$ *lens*, where the get function takes V_i and computes the local view of the node, and the put function that takes a new local view, the current V_i and the extra value E_i and computes a new V_i for a certain node. The E_i supports the put function in obtaining a new state which takes into account a *least-change* criteria.

The second algorithm, for the *Sum of relative deviations* metric, has a different way to process the disagreggation because each node needs as *extra values* the sum of the squared values aggregated from the nodes bellow it. This *extra values* can not be calculated at the same time as the new state of the node is calculated in disagregation, because it can not be derived from the new updated value and the old sum of the squared values, likewise to Algorithm 1. Due to this particularity, the distributed algorithm that supports this metric has an the extra feature that when the disagregation process reaches the leafs of the system (nodes with no nodes below it), it calculates the extra value and starts an aggregation process to update the system concerning to the extra value for the *Sum of relative deviations*. During this process, the algorithm does not allow any update in the system.

This algorithm, has as constant in each node:

- p_i which represents the parent of the node (if the node has parent, e.g., the root node has no parent);

- N_i which is a set that contains the children's id of the node;

As state in each node we have:

- a_i, contains the value stored in the node;

- E_i which contains the squared value of the node value and the sum of the squared values aggregated from the nodes bellow it.

- V_i, is a vector which contains the view of the node (of the value stored in the node) and the local views of its children.

- C_i, is a vector which contains the counter value of the node and of it parent (if it has a parent).

The lenses and functions that are parameters of the algorithm, and instantiated in the table 2, are:

- The *lens* `f`, where the `get` function computes the view from the input change value and the `put` function takes the current local view, the new local view, the current value stored in the node and, as extra argument, the $\otimes(E)$ which computes the extra value of the node, to computes a new input change value concerning to the view;

- $(\!|g, \otimes|\!)$ is an aggregation that computes the news extra values after the disagregation. The $\otimes$ function takes E_i as argument and computes the an aggregated extra value, and `g`, takes the value stored in the node as argument and calculates the new extra value of the node.
  To not allow *output changes* before the conclusion of the disagregation and aggregation process, the disaggregation process set the extra values (E_i) to $null$ and thus the $\otimes$ function returns $\perp$ if the node does not contain all the extra values. In the aggregation process, the `g` function assign the new values to E_i, and when the aggregation finishes the system is ready to receive *output changes*.

Algorithm 1: Bidirectional distributed data aggregation algorithm

constants:
  p_i, parent of node
  N_i, set of children
  E_i, extra values to support in calculating the new local view
  f, is a lens that computes the view from the input value of each node.
  $\oplus$, is a lens between $\vec{V}$ and v to aggregate the local view. The put receives as extra argument E_i

state:
  a_i, input value of the node
  V_i, local view of the node
  C_i, versions of each connection: initially, $C_i = \{i \mapsto 0, p_i \mapsto 0\}$

on $\mathsf{inputChange}_i(value)$
  $a_i := value$
  $V_i(i) := \mathsf{get}_{\mathsf{f}}(a_i)$
  if p_i **then**
    $\mathsf{send}_{i,p_i}(\mathsf{get}_{\oplus}(V_i), C_i(p_i))$

on $\mathsf{outputChange}_i(value)$
  $\mathsf{update}(value)$
  if p_i **then**
    $\mathsf{send}_{i,p_i}(\mathsf{get}_{\oplus}(V_i), C_i(p_i))$

on $\mathsf{receive}_{j,i}(value, version)$
  if $j = p_i$ **then**
    $C_i(p_i) := version$
    $\mathsf{update}(value)$
  else
    if $C_i(i) = version$ **then**
      $V_i(j) := value$
      if p_i **then**
        $\mathsf{send}_{i,p_i}(\mathsf{get}_{\oplus}(V_i), C_i(p_i))$

procedure $\mathsf{update}(value)$
  $V_i := \mathsf{put}_{\oplus}(V_i, E_i, value)$
  $a_i = \mathsf{put}_f(V_i(i), a_i)$
  $C_i(i) := C_i(i) + 1$
  foreach $j \in N_i$ **do**
    $\mathsf{send}_{i,j}(V_i(j), C_i(i))$

Algorithm 2: Bidirectional distributed data aggregation algorithm

constants:
- p_i, parent of node
- N_i, set of children
- f, is a lens that computes the view from the input value of each node and the put receives as extra argument computes the new input value
- $\oplus : \vec{V} \to v$, computes a value which represents the view of the node.
- $(\!|g, \otimes|\!)$ is an aggregation that computes the extra values.

state:
- a_i, input value of the node
- E_i, extra values to support in calculating the new local view
- V_i, local view of the node
- C_i, versions of each connection: initially, $C_i = \{i \mapsto 0, p_i \mapsto 0\}$

on $\mathsf{inputChange}_i(value)$
- $a_i := value$
- $E_i(i) := \mathsf{get_g}(a_i)$
- $V_i(i) := \mathsf{get_f}(a_i)$
- **if** $p_i \wedge \otimes(E_i) \neq \bot$ **then**
  - $\mathsf{send}_{i,p_i}(\oplus(V_i), \otimes(E_i), C_i(p_i))$

on $\mathsf{outputChange}_i(value)$ **where** $\otimes(E_i) \neq \bot$
- $\mathsf{update}(\oplus(V_i), value, \otimes(E_i))$

on $\mathsf{receive}_{j,i}(v, value, e, version)$
- $C_i(p_i) := version$
- $\mathsf{update}(v, value, e)$

on $\mathsf{receive}_{j,i}(v, e, version)$
- **if** $C_i(i) = version$ **then**
  - $V_i(i) := \mathsf{get_f}(a_i)$
  - $E_i(i) := \mathsf{get_g}(a_i)$
  - $V_i(j) := v$
  - $E_i(j) := e$
  - **if** $p_i \wedge \otimes(E_i) \neq \bot$ **then**
    - $\mathsf{send}_{i,p_i}(\oplus(V_i), \otimes(E_i), C_i(p_i))$

procedure $\mathsf{update}(v, value, e)$
- $a_i := \mathsf{put_f}(v, value, e, a_i)$
- $C_i(i) := C_i(i) + 1$
- $E_i := \{k \mapsto \bot \mid (k, \_) \in E_i\}$
- **if** $N_i = \emptyset$ **then**
  - $V_i(i) := \mathsf{get_f}(a_i)$
  - $E_i(i) := \mathsf{get_g}(a_i)$
  - **if** p_i **then**
    - $\mathsf{send}_{i,p_i}(\oplus(V_i), \otimes(E_i), C_i(p_i))$
- **else**
  - **foreach** $j \in N_i$ **do**
    - $\mathsf{send}_{i,j}(v, value, e, C_i(i))$

	Sum	Min

Δ^{ssd}	$\mathsf{get_f}(a) := b$
	$\mathsf{put_f}(b,a) := a$
	$\mathsf{get}_{\oplus}(V) := \sum V$
	$\mathsf{put}_{\oplus}(V,E,v) :=$
	$V + (v - \sum V) \times \frac{E(i)}{\sum E}$
		$\mathsf{get_f}(a) := b$
		$\mathsf{put_f}(b,a) := a$
		$\mathsf{get}_{\oplus}(V,E,v) := \mathsf{Min}\, V$
		$\mathsf{put}_{\oplus}(V,E,v) :=$
		if $v < \mathsf{Min}\, V$ **then**
		$V\{k \mapsto v\}$, **where** $(k, \mathsf{Min}\, V) \in V$
		else
		$\{k \mapsto Max(v, V(k)) \mid (k,v) \in V\}$

Δ^{cn}	$\mathsf{get_f}(a) := b$
	$\mathsf{put_f}(b,a) := a$
	$\mathsf{get}_{\oplus}(V) := \sum V$
	$\mathsf{put}_{\oplus}(V,E,v) :=$
	$V\{i \mapsto V(i) + v - \sum V\}$
		$\mathsf{get_f}(a) := b$
		$\mathsf{put_f}(b,a) := a$
		$\mathsf{get}_{\oplus}(V,E,v) := \mathsf{Min}\, V$
		$\mathsf{put}_{\oplus}(V,E,v) :=$
		if $v < \mathsf{Min}\, V$ **then**
		$V\{i \mapsto v\}$, **where** $(k, \mathsf{Min}\, V) \in V$
		else
		$\{k \mapsto Max(v, V(k)) \mid (k,v) \in V\}$

Table 1.: Algorithm 1 functions

	Sum
Δ^{srd}	$\mathsf{get_f}(a) := b$ $\mathsf{put_f}(v,v',e,a) := a + \frac{a^2}{e} \times (v' - v)$ $\mathsf{g}(a) := a^2$ $\oplus(V) := \sum V$ $\otimes(E) := \sum E$

Table 2.: Algorithm 2 functions

5

SIMULATOR

This tool simulates in a distributed environment the bi-direccionalization of some aggregation primitives (SUM and MIN) using the "Discrete Event Simulation" model of simulation. It is assumed that the system has tree topology, no message loss, asynchronous communication, bidirectional paths to connect nodes, each node has a value that will be aggregated in the root, and updates can be performed in any node (but it only affects the values of the children).

Being the aggregation primitives bidirectional, after aggregating the existing values in the system there is the possibility to assign a new state to the system (e.g.: the SUM of the system is 70) - through the disaggregation. Thus, on disaggregation, the new state is assigned to the system through a distributed process in which the decisions are taken node-to-node, based on the node's local vision about the system.

This simulator recreates the two algorithm represented in Section 4.6 but due to its specification and needs concerning to the extra values (E_i), each algorithm is simulated at a time. For each aggregation primitive and least-change criteria an instantiation of the simulator is also defined.

Besides the simulation of aggregation and disaggregation of primitives, the simulator has two types of verification: on the first one the system is assumed to be a multiset with all existing nodes and after the disaggregation we verify if the system has one of the correct possible states (relative to the node's value), i.e. we check the correctness of the lens; the second verification is to check, after the simulation if the system converged to a least-change solution.

The main decisions about the design of the simulator and how this decisions were implemented, will now be described.

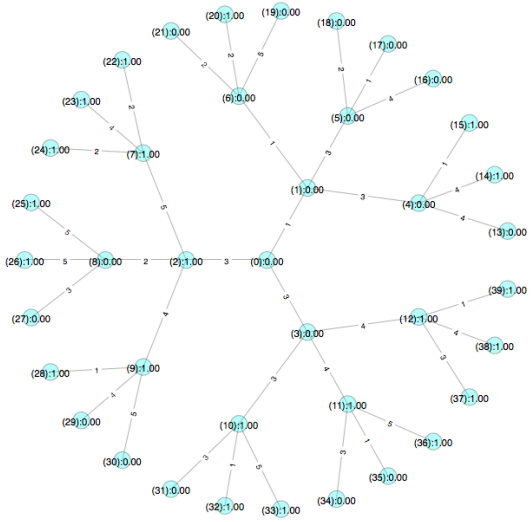


Figure 14.: System overview

5.1 SIMULATION

When the simulator is initiated, it will generate the system with a given branching factor of the tree and the height factor of the tree, randomly generates the node's value between a given interval and the message delay for each edge, and an inputChange is performed to each *leaf* node in the system - so that all nodes have a local vision of the system.

The simulator has the option to visualize the system state - topology, node's values and edge's delays (Figure 14), to simulate crossed requests, unit requests and a set of randomly generated requests, and finally, the option to verify if the system is correct concerning to the performed requests.

5.2 VERIFICATION

The verification feature has as main purpose to check the system state after the disaggregation. Thus, the user can perform two types of verification: *Correctness verification* and *Least-change verification*.

5.2.1 *Correctness verification*

On correctness verifications, the system is assumed as a multiset with all nodes of the system and each node is represented by a pair (id-value) and the request to change the system state will affect all nodes. In practical terms the requests will always be performed in the node that represents the root of the system - since the purpose is to evaluate the algorithm's behavior and it is not necessary to consider

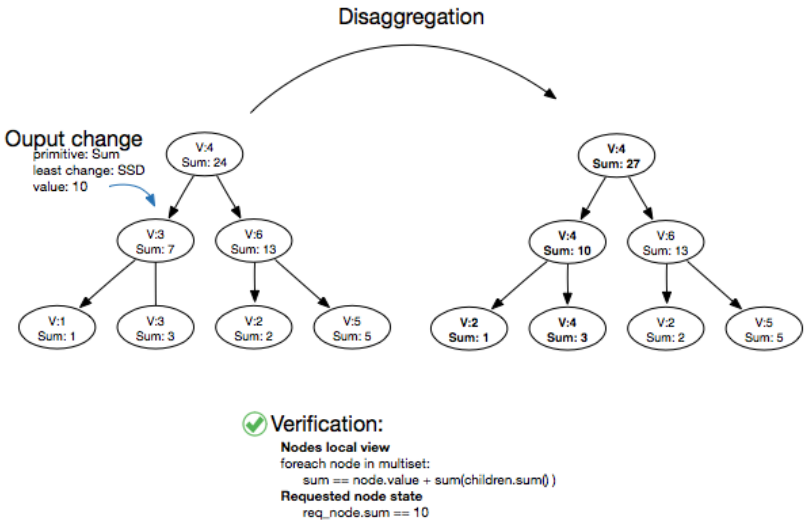


Figure 15.: Correctness verification

the topology.

Before the simulation (disaggregation) the system state is logged and the root node of the system is updated (where the request will be performed). After the simulation the updated system state is logged and verified if it is correct concerning to the initial system state and to the performed request (Figure 15).

5.2.2 *Least-change verification*

The least-change verification has as main goal to verify if the system after the disaggregation converges to a correct state (the aggregated values in each node are correct) and if the requested was performed successfully (e.g.: if the request is ”change sum to N in node M”, after the disaggregation the sum logged in the node M should be N).

Before the simulation (disaggregation) only the node values are logged, and after the disaggregation it is verified in each node if its value is correct concerning to the value requested (Figure 16).

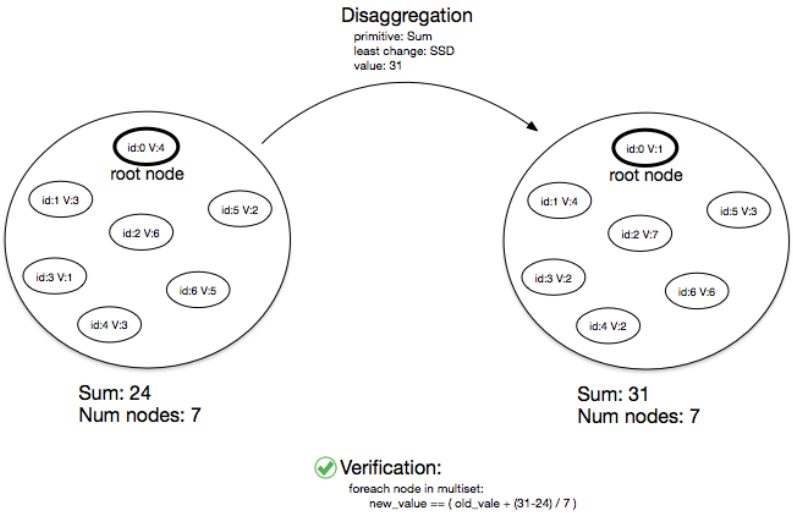


Figure 16.: Least-change verification

5.3 DESIGN AND IMPLEMENTATION

5.3.1 *Design*

Being this tool designed for extensibility, the main design decisions take into account the division of the main subjects (distributed algorithm, primitives, least changes) for a better maintenance and evolution of the simulator.

Compositional approach

The simulator has as main goal to simulate a distributed algorithm that processes the aggregation and the disaggregation of primitives, and in disaggregation there are measures (least changes) associated to the primitives (e.g.: perform SUM primitive with ”Sum of Squared Differences” least change, or perform MAX primitive with ”Sum of Squared Differences” least change), which are common to all primitives. Therefore, the simulator was designed with a compositional approach to separate the part of the algorithm that represents the distributed logic form the part that represents the instantiation of the primitives and least change criteria.

The structure of the simulator is presented in Figure 17.

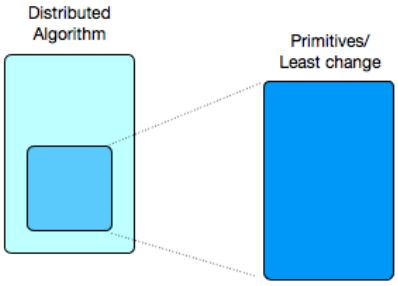


Figure 17.: Composicional approach in implementation

Error deviation

Due to rounding of values, there is a tolerance when the simulator compares values - in simulation or in verification. To obtain a reliable comparison concerning to the compared values, the relative difference between the values is computed and we verify if this difference is less or equal to a given value (tolerance). The defined value to tolerance is 0.001, and if the relative difference is greater than this value, the compared values are considered different.

If both compared values have an absolute difference from zero less or equal than 0.0001, they are considered equal.

Node information

In the disaggregation process the algorithm goes through the nodes making changes, taking decisions concerning to the local view of the node. To achieve this, each node has to contain information to support the algorithm to make the best decisions concerning to the primitive and the least change. To accomplish this, each node should have the following information:

- Node parent (p_i)
- Node value (a_i)
- Set with the children's ID (N_i)
- Collection with its aggregated value and the children's aggregated value (V_i)
- Collection with extra values (its extra value and children's extra value) to support the calculation of the new views (E_i)
- Collection with its and parent counter value C_i

5.3.2 *Implementation*

This simulator was implemented in Python 2.7. It was based on the ScyPy, an ecosystem of open-source software for mathematics, science, and engineering - to support the representation of the system in a structural and graphic level.

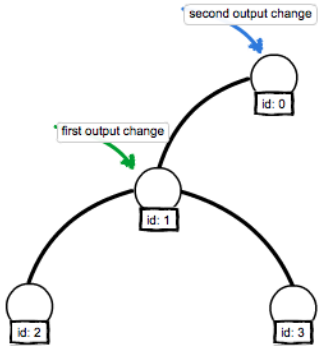
The different features of the simulator were divided through the following classes - following the structure presented before in the Design.

- `dis_event_sim.py` class coordinates "Discrete Event Simulation".
- `event.py` class represents the object which is handled by the *dis_event_sim. py* and contains information about the event (e.g., if is an outputChange or an inputChange, the counter value, ...)
- `node.py` class represents each node in the system and keeps the local vision of the system.
- `functions.py` class is responsible to perform the existing primitives and least-change metrics (f, g, $\oplus$, $\otimes$).
- `tests.py` class verifies the system after the disaggregation (Correctness test and least-change verification)
- `utils.py` class has auxiliary functions to support the simulation (e.g., `is_equal` function which implements the *Error deviation* 5.3.1).

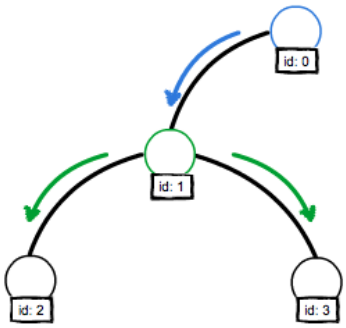
5.4 CONCURRENT UPDATES ON ALGORITHM 2

The second algorithm has the limitation of not supporting concurrent updates (when a second update is performed in a node above the node where the first update was performed), as Figure 18 illustrates. This limitation occurs because in the disaggregation process the algorithm calculates a new input change (a_i) for the nodes (figure 18b), and this new value is calculated taking into account the current input change value of the node and the local view of the node where the *output change* was performed (algorithm 2, line 32). Having concurrent updates, the disaggregation process that corresponds to the second update will calculates the new input change values based on values which do not correspond to the local view of the node where the second update was requested (figure 18c and 18d), thus the final state of the system will not be the requested by the second update (the update which should win), as shown in Figure 18e.

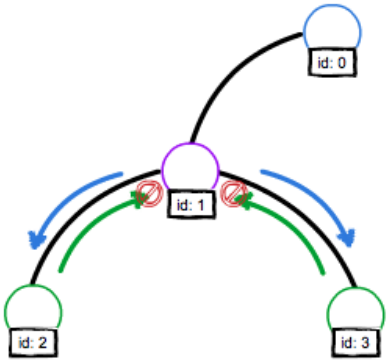
This limitation was identified with the help of the developed simulator when the second algorithm was tested in scenarios with concurrent updates.



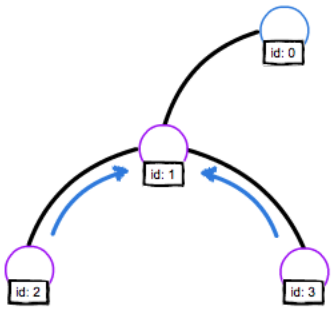
(a) Two *output changes*, at the same time, in different nodes from the same sub-tree



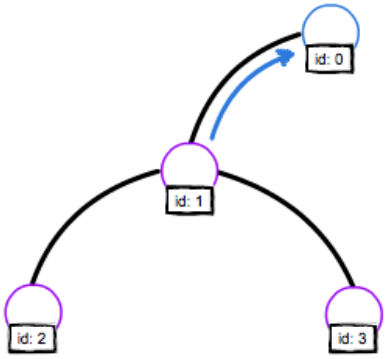
(b) The updates and information propagation through the sub-tree



(c) Incorrect update in node 1 by the second *output change*. The first *output change* is stopped due to the versions.



(d) The second *output change* continues calculating wrong values through the system.



(e) The final (incorrect) state

Figure 18.: Algorithm 2, concurrent updates

6

CONCLUSION

Bidirectional transformations has a enormous interest and importance due to its enormous applicability in various domains, and its application in a distributed settings, subject poorly addressed, could bring useful solutions for realistic and important scenarios.

In this master thesis we present an introduction to the combination of bidirectionality with distributed aggregation, by combining existing theory in these two fields to achieve an unique approach. First we choose realistic scenarios and tried to understand how this approach would be useful. This lead us to the introduction of *least change* metrics in the disaggregation process, i.e., criteria to control how the bidirectionalization is achieved in a distributed system.

With the experimental results, obtained by using the developed simulator, was identified the major obstacles when combining these two subjects, namely how to choose which distributed algorithm is better suited for each aggregation primitive and leas-change criteria, the importance of the metric be self-decomposable, and how causality plays an important role in this approach.

There is a lot of future work to do in this new approach, such as the construction of a more practical solution to apply when in a disaggregation the new local view of the node can not be derived from the updated, or the implementation of invariants. Although these ambitious challenges are left for future work, we believe we already have a solid first step towards an effective bidirectional distributed data aggregation algorithm.

BIBLIOGRAPHY

J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3), May 2007. ISSN 0164-0925. doi: 10.1145/1232420.1232424. URL http://doi.acm.org/10.1145/1232420.1232424.

Paulo Jesus, Carlos Baquero, and Paulo Sérgio Almeida. A survey of distributed data aggregation algorithms. *CoRR*, abs/1110.0725, 2011.

Nuno Macedo, Hugo Pacheco, Alcino Cunha, and José Nuno Oliveira. Composing least-change lenses. *ECEASST*, 57, 2013.

Lambert Meertens. Designing constraint maintainers for user interaction. In *Third Workshop on Programmable Structured Documents*. Tokyo University, 2005.

Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, pages 124–144, New York, NY, USA, 1991. Springer-Verlag New York, Inc. ISBN 0-387-54396-1. URL http://dl.acm.org/citation.cfm?id=127960.128035.

Hugo Pacheco, Nuno Macedo, Alcino Cunha, and Janis Voigtländer. A generic scheme and properties of bidirectional transformations. *arXiv preprint arXiv:1306.4473*, 2013.

Robbert Van Renesse. The importance of aggregation. pages 87–92, 2003.

Part III

APENDICES

A

OPTIMAL SOLUTION TO MINIMIZE THE SUM OF RELATIVE DEVIATION

A proof of how to get the best solution to change the sum, in the root node, from 10 to 15 with the minimum sum of relative deviations.

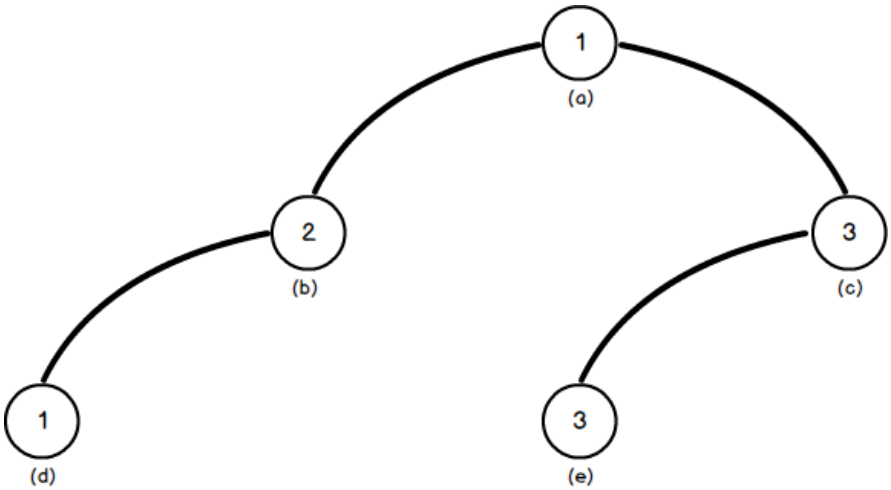


Figure 19.: Distributed system

Info	Value
Sum (root node)	10
Update to	15
Difference	$15 - 10 = 5$

Table 3.: Test data

We want to minimize the followimg function:

$$f(m) = \sum_{id} \left(\frac{m_i' - m_i}{m_i} \right)^2 \tag{33}$$

$$f(m) = \sum_{id} \left(\frac{m'_i - m_i}{m_i} \right)^2 \Leftrightarrow f(d,m) = \sum_{id} \left(\frac{d_i}{m_i} \right)^2 \tag{34}$$

$$f(d,m) = \left(\frac{d_a}{1} \right)^2 + \left(\frac{d_b}{2} \right)^2 + \left(\frac{d_c}{3} \right)^2 + \left(\frac{d_d}{1} \right)^2 + \left(\frac{d_e}{3} \right)^2 \tag{35}$$

Subject to the following constraint (the sum of the differences in each node is equal to 5):

$$g(d) = c \Leftrightarrow \sum_{id} d_i = 5 \tag{36}$$

Lagrange multipliers

The proof is presented by using the Lagrange Multpliers technique:

$$\Lambda(d,\lambda) = f(d) + \lambda \left(g(d) - c \right) \tag{37}$$

the partial derivatives

$$\begin{aligned}
\frac{\partial}{\partial d_a}\Lambda(d,\lambda) &= 2d_0 + \lambda \\
\frac{\partial}{\partial d_1}\Lambda(d,\lambda) &= \frac{1}{2}d_1 + \lambda \\
\frac{\partial}{\partial d_2}\Lambda(d,\lambda) &= \frac{2}{9}d_2 + \lambda \\
\frac{\partial}{\partial d_3}\Lambda(d,\lambda) &= 2d_3 + \lambda \\
\frac{\partial}{\partial d_4}\Lambda(d,\lambda) &= \frac{2}{9}d_4 + \lambda \\
\frac{\partial}{\partial \lambda}\Lambda(d,\lambda) &= d_a + d_b + d_c + d_d + d_e - 5 = 0
\end{aligned}$$

Solving the system

$$l(d,\lambda) = \begin{cases} 2d_a + \lambda = 0 & d_a = -\frac{1}{2}\lambda \\ \frac{1}{2}d_b + \lambda = 0 & d_b = -2\lambda \\ \frac{2}{9}d_c + \lambda = 0 & d_c = -\frac{9}{2}\lambda \\ 2d_d + \lambda = 0 & \Leftrightarrow \quad d_d = -\frac{1}{2}\lambda \\ \frac{2}{9}d_e + \lambda = 0 & d_e = -\frac{9}{2}\lambda \\ d_a + d_b + d_c + d_d + d_e - 5 = 0 & d_a + d_b + d_c + d_d + d_e - 5 = 0 \end{cases}$$

substitute d_a, d_b, d_c, d_d and d_e in the sixth equation

$$l(d,\lambda) = \begin{cases} d_a = -\frac{1}{2}\lambda & d_a = -\frac{1}{2}\lambda \\ d_b = -2\lambda & d_b = -2\lambda \\ d_c = -\frac{9}{2}\lambda & d_c = -\frac{9}{2}\lambda \\ d_d = -\frac{1}{2}\lambda & \Leftrightarrow \quad d_d = -\frac{1}{2}\lambda \\ d_e = -\frac{9}{2}\lambda & d_e = -\frac{9}{2}\lambda \\ -\frac{1}{2}\lambda - 2\lambda - \frac{9}{2}\lambda - \frac{1}{2}\lambda - \frac{9}{2}\lambda - 5 = 0 & \lambda = -\frac{10}{24} \end{cases}$$

substitute λ in the first, second, third, fourth and fifth equation and obtain the solution

$$l(d,\lambda) = \begin{cases} d_a = -\frac{1}{2} * \left(-\frac{10}{24}\right) & d_a = \frac{10}{48} \\ d_b = -2 * \left(-\frac{10}{24}\right) & d_b = \frac{20}{24} \\ d_c = -\frac{9}{2} * \left(-\frac{10}{24}\right) & d_c = \frac{90}{48} \\ d_d = -\frac{1}{2} * \left(-\frac{10}{24}\right) & \Leftrightarrow \quad d_d = \frac{10}{48} \\ d_e = -\frac{9}{2} * \left(-\frac{10}{24}\right) & d_e = \frac{90}{48} \\ \lambda = -\frac{10}{24} & \lambda = -\frac{10}{24} \end{cases}$$

Function to distribute the changes

$$d_i = \frac{m_i^2}{\sum_{id} m_i^2} * dif$$