Visualizing Project Relations on GitHub

Hendrik van Antwerpen
H.vanAntwerpen@student.tudelft.nl

Niels van Kaam
N.vanKaam@student.tudelft.nl

Software Engineering Research Group, EECMS
Delft University of Technology

Abstract

GitHub has emerged as an interesting object of study for software repository analysis because of the data available through its open API. Visualizing this data can be challenging, because of the lack of knowledge about its structure and because of the size of the dataset. For distributed processing of large data sets functional programming languages and techniques and models like MapReduce have become increasingly popular. In this report we describe a web based system that makes it possible to explore one type of relation in GitHub, the links between projects based on common committers. Users can explore the data by selecting time limits, minimum link weight and project language. This report describes the implementation of that system in the object-oriented function programming language Scala, using Sawzall style aggregators and other function programming techniques.

1 Introduction

In the field of software repository mining, GitHub has emerged as an interesting subject of study and tools have emerged to collect and publish the data it publishes (Gousios and Spinellis, 2012). Making sense of the data and visualizing it in interesting ways is not trivial. Finding interesting correlations and metrics is hard. Interactive software that visualizes aspects of the data while the user can influence constraints like time period or project language allows easy exploration of the data and can be a starting point for more formal analysis.

We present a web based software system that allows a user to explore relations between projects on GitHub. A relation is defined by the existence of a common committer. The user can influence the time period wherein the link must exists, the main programming language of the project and the minimum number of committers before a link is shown.

We identify several challenges in the design of the software:

- Providing a general visualization that might reveal interesting properties of the data to the user.
- Dealing with the large size of the data that is being processed.
- Ensuring responsiveness and reduce waiting times of the software to encourage the user to explore.

#projects	600k
#users	400k
#commits	14m
#unique project-user links per week	3m
#project links	13m

Table 1: Some numbers about the data set

The software was developed in the context of a functional programming course[1] at the Delft University of Technology. The course focused on functional programming techniques, application of these techniques in more imperative languages and using them for processing large data sets. The report therefore also describes how the functional programming and data processing techniques from the course were applied. The project was implemented using Scala and the code samples will be mostly in that language.

The report is structured as following:

- A description of the data and the data processing (section 2).
- A description of a MapReduce implementation in Scala, created to implement the data processing (section 3).
- A description of the distributed design of the backend, created to improve responsiveness of the software (section 4).
- A description of the visualization used in the software (section 5).
- Conclusions and ideas for further research and development (section 6).

2 Data set and processing strategy

The raw data available was a list of projects and their main programming language, a list of users and a list of commits. A commit provides the basic link between a user and a project at a certain time. A link between projects exists when the have a common committer. We define the weight of a link to be the number of common committers and the weight of a project to be its degree, i.e. the number of projects it is connected to. Table 1 lists some numbers on the data set. The distribution of unique project-user links per week is shown in figure 1. Figure 2 shows a histogram of link weight. A histogram of the number of projects per user is shown in figure 3.

We allow the user to vary the following parameters:

- Project language (limits projects within one ecosystem).
- Time period in which the links are counted.
- Minimum weight of a link before it's shown.

The desired end result is a list of links between projects, a weight on each link of the amount of common committers and a weight on each project of the amount

[1] `http://swerl.tudelft.nl/bin/view/Main/FunctionalProgrammingCourse`

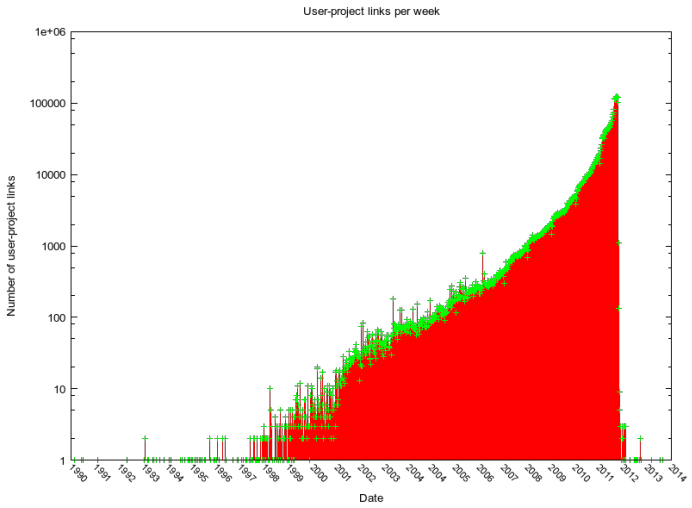

Figure 1: Unique user-project links by week

of connected projects. This last property can identify central projects in the graph.

Processing data can be elegantly expressed as a stream of filter and transform operations. Without considering implementation details, those steps will be:

1. Filter projects for selected language.
2. Filter commits and exclude commits outside of the selected time period or not in the filtered project list.
3. Build a set of projects per user.
4. For each user's projects, create links between those projects.
5. Merge project links, assigning them weight based on their occurrence.
6. Transform the list of links to a format understood by the visualization code.

3 MapReduce with Scala collections

An increasingly popular model for processing big data sets is MapReduce (Dean and Ghemawat, 2008). This model lends itself well for distribution and parallel data processing. One descendant of this model is found in Google's Sawzall (Pike et al., 2005), using aggregators as a key abstraction. In Lämmel (2008) a rigorous description of these two models is provided. The Sawzall concept of aggregators is identified as a generalized form of monoids. Their properties allow easy combination of parallel computed partial results, possible in a hierarchical way. Because monoids for common data types like lists, maps and tuples are readily available, developers need think less about the reduction of intermediate results and can focus on the map part of the process.

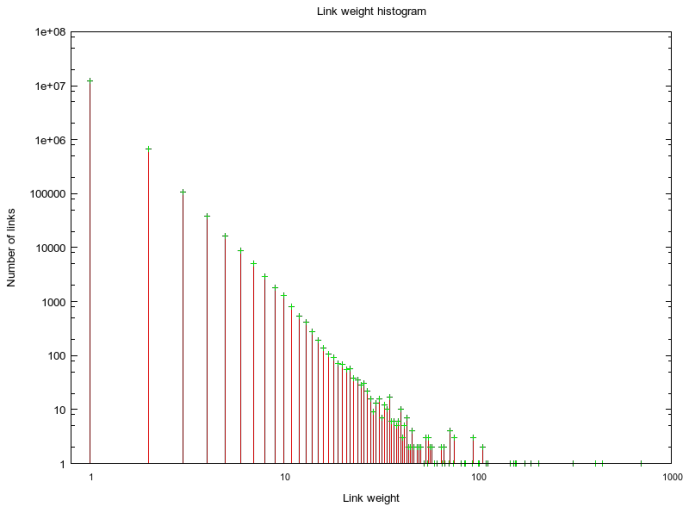

Figure 2: Link weight histogram

Sawzall is very liberal in the output it accepts from map functions. The output can be a monoid, a collection of the monoid or an element if the monoid is a collection itself. To capture all these Lämmel defines aggregators as generalized monoids:

```
1  class Monoid m => Aggregator e m | e -> m
2    where
3      minsert :: e -> m -> m
```

To our knowledge no implementation of this approach is available in Scala. Therefore we implemented a version of aggregators for Scala and used those to implement a map-reduce function that integrates seamless with the Scala collection library.

3.1 Aggregators

To implement the Sawzall model, we will start with the key abstraction, the aggregator. An aggregator is defined by the following trait:

```
1  trait Aggregator[A,B] {
2    def zero: A
3    def insert(a: A, b: B): A
4  }
```

Because there is no readily available monoid type in Scala and we will use aggregators only, we dropped the requirement that `A` has to be a monoid. We use the term monoid aggregator for any `Aggregator` where `A` and `B` are the same type, because this behaves the same as a monoid of that type would.

We defined the `minsert` function in Scala as `a:A |<| b:B : A`. Whenever this operator is used, the compiler infers which `Aggregator[A,B]` to use. All Scala **`implicit`** rules apply, so if no `Aggregator` is found there's compile time error, if multiple are found the programmer has to specify programmatically which one to use.

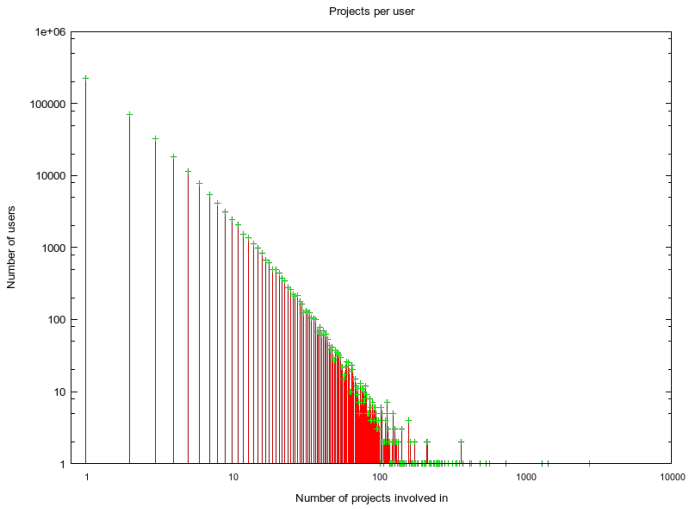

Figure 3: Projects per user histogram

We started with a basic monoid aggregator for sums, which serves as the default aggregator for integers in the rest of the examples. The implementation looks like:

```
1  implicit def SumMonoid =
2    new Aggregator[Int,Int] {
3      override def zero = 0
4      override def insert(a: Int, b: Int) = a + b
5    }
```

The next step is implementing collection aggregators. Generally every collection type has two aggregators. One for the monoid case and one for the element insert case. For the case of lists, the signatures look like

```
1  implicit def ListMonoid: Aggregator[List[A],List[A]] = ...
2  implicit def ListAggregator: Aggregator[List[A],A] = ...
```

Implementing this for a lot of collection types would be a daunting task. Luckily the design of the generic Scala collections library (Odersky and Moors, 2009) comes to great help here. By using the `CanBuildFrom[Coll,Elem,Result]` infrastructure and higher-order generics, we only have to implement two aggregators for any kind of `GenSet`, for any kind of `GenSeq` and for any kind of `GenMap`.

As an example, lets look at the aggregators for `GenSeq`, shown in listing 1. The two cases (monoid and element) are implemented separately. Although the implementation is relatively straightforward, some things are note-worthy. In the monoid case, we are more liberal in our input than the type we produce. Every collection implements the `GenTraversableOnce[A]` interface for it's element types and allows easy appending of such a collection. Apart from the case `List[Int] |<| List[Int]` this also allows us to do things like `List[Int] |<| Set[Int]`. Apart from being liberal in the collection type we insert, we are also covariant in the collections element type. This is intuitive for programmers, when a collection takes elements of type `A`, one can also insert elements of a subtype `B <: A`. Because type inferences considers the type `Elem` to be invariant, we added the extra `InElem <:Elem` to allow any subtype as well.

Listing 1: Aggregators for sequences

```
implicit def GenSeqAggregator[Repr[X] <: GenSeq[X], Elem, InElem <:
    Elem]
                              (implicit bf: CanBuildFrom[Nothing,Elem,
                                  Repr[Elem]]) =
  new Aggregator[Repr[Elem],InElem] {
    override def zero: Repr[Elem] = bf().result
    override def insert(s: Repr[Elem], e: InElem) =
      (s :+ e).asInstanceOf[Repr[Elem]]
  }

implicit def GenSeqMonoid[Repr[X] <: GenSeq[X], Elem, In[X] <:
    GenTraversableOnce[X], InElem <: Elem]
                         (implicit bf: CanBuildFrom[GenSeq[Elem],Elem,
                             Repr[Elem]]) =
  new Aggregator[Repr[Elem],In[InElem]] {
    override def zero: Repr[Elem] = bf().result
    override def insert(s1: Repr[Elem], s2: In[InElem]) =
      (s1.++(s2)(bf))
  }
```

For `Map` there was some extra work to be done. Although `Map[K,V] <:Traversable[(K,V)]` the compile would not infer the aggregator when a map was provided, like `Map[Int,Int] |<| Map[Int,Int]`. Therefore another monoid aggregator was implemented for this specific case, although still general for any `Map` type.

This brings the count for our collection aggregators to seven and covers all cases. Some examples to show what we can do with it:

```
// Int |<| Int : Int
1 |<| 2 // = 3

// List[Int] |<| Set[Int] : List[Int]
List(1,2) |<| Set(3) // = List(1,2,3)

// SortedSet[Int] |<| Int : SortedSet[Int]
SortedSet(2,3) |<| 1 // = SortedSet(1,2,3)

// simple word count coming up!
// Map[String,Int] |<| (String,Int) : Map[String,Int]
Map("aap"->1,"noot"->2) |<| ("noot",1) // = Map("aap"->1,"noot"->3)
```

A nice property of monoids is that they can be structurally combined. For example a tuple with two monoid values is itself a monoid. In our aggregator world, we happen to have similar properties. Aggregators for `TupleN` (see listing 2) and `Map` where implemented to require their tuple values and map value to be aggregators themselves. This is a difference with Lämmels description, where the nested values are strict monoids. Our approach introduces a lot of flexibility in the elements that can be inserted. When creating deep structures, on every level we have the choice to insert either a value or a collection. Some examples should show the flexibility this gives us:

```
// count individual and total words
// (Int,Map[String,Int]) |<| (Int,(String,Int)) : (Int,Map[String,Int])
(3,Map("aap"->1,"noot"->2)) |<| (1,("aap",1)) // = (4,Map("aap"->2,"
    noot"->2))

```

Listing 2: Aggregator for tuple

```
implicit def Tuple2Aggregator[A1,A2,B1,B2]
             (implicit a1: Aggregator[A1,B1],
                       a2: Aggregator[A2,B2]) =
  new Aggregator[(A1,A2),(B1,B2)] {
    override def zero = (a1.zero,a2.zero)
    override def insert(a: (A1,A2), b: (B1,B2)) =
      (a1.insert(a._1, b._1),a2.insert(a._2, b._2))
  }
```

Listing 3: Aggregator for collections of elements

```
implicit def GroupAggregator[Coll,In[X] <: GenTraversableOnce[X],Elem]
                            (implicit va: Aggregator[Coll,Elem])=
  new Aggregator[Coll,In[Elem]] {
    override def zero = va.zero
    override def insert(a: Coll, as: In[Elem]) =
      (a /: as)( (c,e) => va insert (c,e) )
  }
```

```
// or (Int,Map[String,Int]) |<| (Int,Map[String,Int]) : (Int,Map[String
    ,Int])
(1,Map("aap"->1)) |<| (3,Map("aap"->1,"noot"->2)) // = (4,Map("aap
    "->2,"noot"->2))
```

One aspect of the Sawzall aggregators is not addressed yet: the possibility to insert a collection of elements. For this case we implemented an aggregator similar to Lämmel's `Group` aggregator, shown in listing 3. Note that this works again at every level of nested types, so this allows us to do things like:

```
// Int |<| List[Int] : Int
3 |<| List(2,5) // = 10

// Map[Int,Int] |<| List[(Int,List[Int])] : Map[Int,Int]
Map(1->1) |<| List((1,List(2,3)),(7,List(42))) // = Map(1->6,7->42)
```

Our design allows a lot of freedom in the shape of the elements inserted into a collection. What type of elements can be inserted is dictated by the defined aggregators and fully inferred by the compiler. The aggregators for the Scala collections are very generic and in most cases the developer doesn't have to care about how to merge collections. Monoid aggregators are implemented for string concatenation and integer summing, but others are very easy to implement; one implicit function and an implementation of the `Aggregator` trait is enough.

3.2 MapReduce

Using the developed aggregators, a map-reduce library was implemented. We want the map-reduce functionality to be as easy to use as the standard collection functions, like `map` or `filter`. To a high degree this can be achieved by enriching libraries, a process similar to defining type classes in Haskell (see Odersky (2006) for details).

Using again the generic design of the collections library and type inference allows us to write map-reduce by only specifying the expected result type. The

Listing 4: MapReduce for Scala collections

```
object MapReduce {

  class GenMapReducer[Elem](as: GenTraversableOnce[Elem]){
    def mapReduce[ResultColl] = new MapReducer[ResultColl]
    class MapReducer[ResultColl] {
       def apply[ResultElem](f: Elem => ResultElem)
                (implicit p: Aggregator[ResultColl,ResultElem])
                : ResultColl =
             (p.zero /: as)( (c,a) => p.insert(c, f(a)) )
    }
  }

  implicit def mkMapReducable[Elem](as: GenTraversableOnce[Elem]) =
    new GenMapReducer[Elem](as)

}
```

full implementation is shown in listing 4. Our first implementation defined `def mapReduce[ResultColl,ResultElem](f: Elem =>ResultElem)`. Unfortunately it is not possible in Scala to provide some of the type parameters but have others inferred. This forced us to repeat the return type of the function when specifying the result type of the `mapReduce` call. Introducing the intermediate `MapReducer` object solved this problem. The result type of mapReduce was specified on the call, but the return type of the function could be inferred for the `apply` call. Because parentheses can be omitted when a function has no arguments, this resulted in a syntax identical to a case without the intermediate object, but with one less type parameter. A word count example similar to one Lämmel gives now looks like:

```
val docs: List[String] = ...
def wordcount(doc: String): List[(String,Int)] = doc.split(" ").toList.
    map( w => (w,1) )
val wc = docs.mapReduce[Map[String,Int]]( wordcount )
```

The performance of our map-reduce is in the range as hand written code using e.g. `foldLeft`. Most of the work is done through inference by the compiler. At runtime the aggregators use folds and collection operations just as you would in handwritten code. There a little overhead of some function calls, but all the reduction details are abstracted away, resulting in less repetition and simpler data transformation functions.

It is possible that multiple aggregators for the same type exist, for example a sum and a product on numbers. In such a case it is very easy to specify which one to use. Here is a small example:

```
def SumMonoid: Aggregator[Int,Int] = ...
def ProductMonoid: Aggregator[Int,Int] = ...

val words: List[String] = ...
val sumOfWordLengths = words.mapReduce( w => w.size )(SumMonoid)
val prodOfWordLengths = words.mapReduce( w => w.size )(ProductMonoid)
```

As you can see the result type is not required, because it is inferred from the aggregator.

This fully implements the Sawzall map-reduce model as it is formalized in Lämmel (2008). The use of aggregators instead of monoids, gives us even more freedom in the output of our map functions than Lämmel's model does. The integration with the Scala collections library makes using map-reduce very easy for programmers used to that idiom.

With all the pieces in place, we can write our processing steps from section 2 in Scala:

```
1  val commits: List[Commit] = ...
2  def projectsToPairs(ps: List[Project]): List[(Project,Project)] = ...
3  val linkMap =
4    commits.filter( c => c.timestamp >= from && c.timestamp <= until  )
5           .mapReduce[Map[User,Set[Project]]]( c => (c.user,c.project) )
6           .values
7           .mapReduce[Map[(Project,Project),Int]](projectsToParis)
8           .filter( _._2 >= minWeight )
```

4 Distributed data-processing with Akka

Even tough the computation of the project links has been optimised, still a lot of data needs to be processed. Our dataset consisted of 14 million commits. Because of this a request on a set of project links for a given time-interval and weight could take multiple minutes to compute. Such long waiting times are not acceptable, so we decided to distribute the computation by using actors. Because the system was built in Scala, the most logical choice for this is using the Akka toolkit, which is as far as we know the only available actor toolkit for Scala.

4.1 Akka

Akka is a toolkit for, as they state themselves, "building highly concurrent, distributed event driven applications". You can both use akka with Java and Scala, but in our opinion it integrates a lot better with Scala. The main parts in Akka that were used for the distribution were the "Actors" and "Futures". In Akka you can very easily create actors on multiple systems, pass references of actors to other systemsand send messages between actors. When configured properly each actor reference is "network aware", meaning that you can pass a reference from one system to another while the reference remains intact.

With these actors "Futures" become very useful. One can "ask" an actor a specific request, and store the result in a future. A future is a monad, which means operations can de mapped over the future while the result is still being processed by a different actor. Then finally one can pipe the result of the future to another actor, so when the actor which was asked the request fires its result, the operations mapped over the future monad are executed and the result is sent as a message to the actor it was piped to. What Akka actually did behind the scenes was creating another actor, which reacted on the "reply message", computed the operations that were done on the future, and send the result to the actor the future was piped to. This can also be seen in figure 5, which will be explained later.

Is this a correct way of explaining it?

4.2 Distribution

In order to reduce the total computation time by distributing the data over multiple actors, the data needs to be cut into multiple parts. Simply cutting the dataset in multiple pieces is not possible in this case. A link between a project is defined as a "common committer". By splitting the dataset and computing one for example one half on one actor and the other half on a second actor could cause these common commits to be broken.

Therefore we came up with the following solution. We initially give each actor system access to the entire dataset. Then, when an computation-actor is created, it receives a set of users for whose commits it is responsible. This set of users we defined by taking the remainder of a modulo operation on the userid. With a set of user's the actor can now compute the entire process towards the project links, except for the filtering on the weight of the link. For that last operation an actor needs to be sure that it has all links between a project in order to be sure the result is correct, for which it would also the commits from the other users.

A disadvantage is that booting up the system does not have any speedup, or is even slower when reading from the same datasource, because every actor needs to read all commits in order to select the commits based on the users it is responsible for. This however does not influence the response time, because the system is only booted once after which the user's commits are kept in the memory of the computation actors. Finally once the actors respond to the request, they send all the project links they computed to the actor that sent the request (using the "Ask" pattern). The actor that requested the project links receives them and a reduction is done, after which the filter operation over the weight can be done.

The result is the simple system below

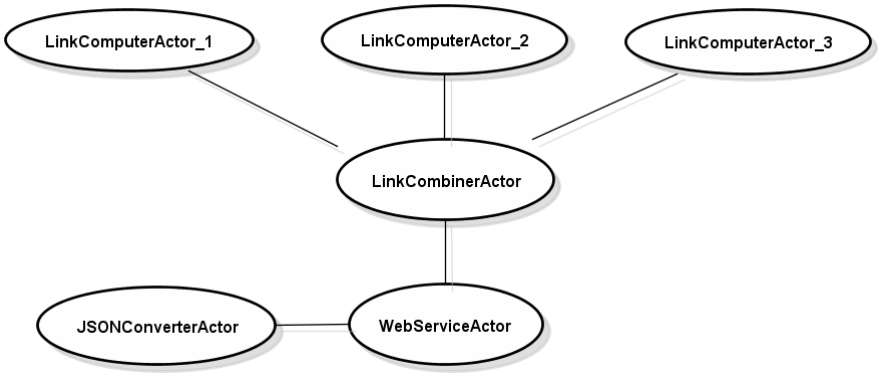

Figure 4: Simple computation actor system

As can be seen in the figure, two more actors were also added in this figure. These are actors that handle communication with the user. The "WebserviceActor" is an actor that receives requests from the user, then creates a future for the LinkResults which are requested from the LinkCombinerActor, and pipes the result to the JSONConverterActor. When the JSONConverterActor receives a list of project links, it will convert them to a JSON format for D3, and send the

JSON code to the user. The entire communication flow when a LinkRequest is done cis shown in figure 5

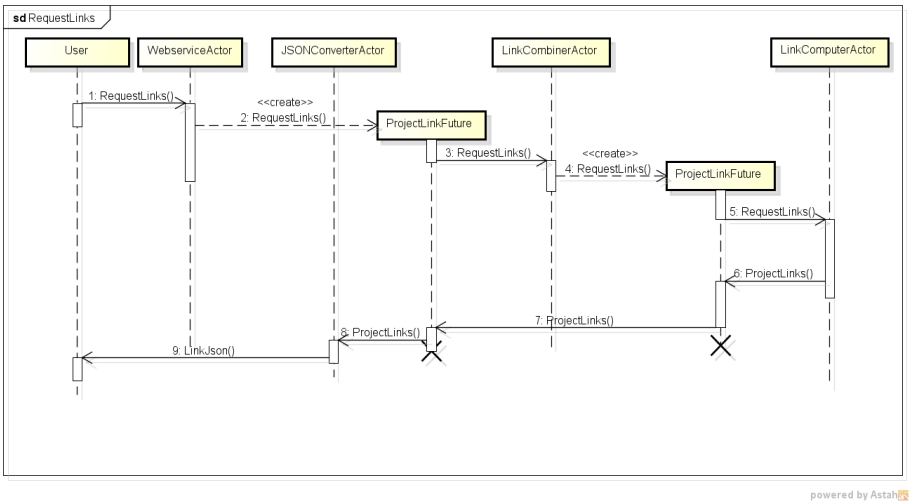

Figure 5: A sequence diagram for a project link request

webservice request ook "asynchroon" in de diagram zetten, dat is ie namelijk wel

When looking at figure 5 it is very nice to see the "WebserviceActor" and the "LinkCombinerActor" only just initiate the process. As described above the result of a request is stored in a future, and the result is piped to a new actor. This means after the initiation the "WebserviceActor" and the "LinkCombinerActor" do need to receive any more messages after that.

4.3 scalability

Naam consequent doen, met of zonder actor in de naam. Met is beter denk ik

A big advantage of the design of the actor system is the scalability. Off course more LinkComputerActors can be used by the single combiner actor in figure 4, but that would mean a lot of project links to be send to the same actor, resulting a lot of data flow at a single actor.

As can be seen in the sequence diagram, both the "LinkCombineActor" and the "LinkComputerActor" can receive a project-link request. Because the actors are completely independent, they do not need to know how the project links are obtained. Therefore both actors can be substituted for each other, as shown in figure 6. This way a tree can be constructed with link combiners and link computers, and each link computer will reduce the project-links of the next level in the tree. This way the system can be scaled into any desired size.

beter uitleggen dat het proces eventdriven is en niet blockt.

4.4 speedup

When computing all project links for a single month on the same computer, with average hardware, it takes about 60 seconds to compute all project links.

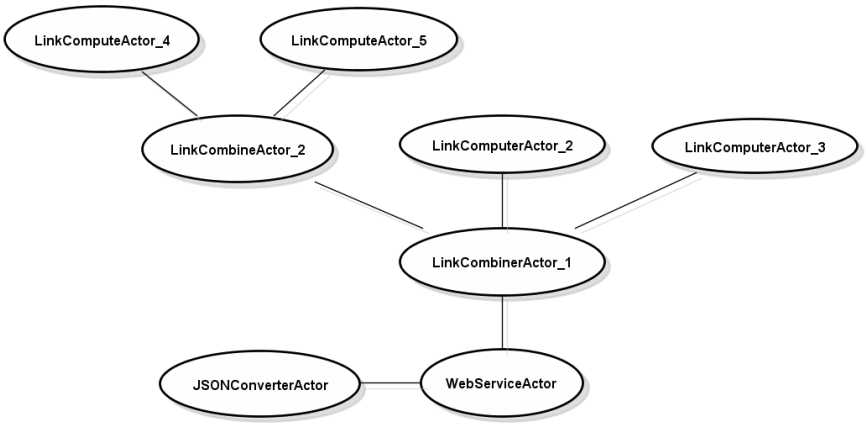

Figure 6: Example of extended actor system

Below here, the runtime stats per step, + explanation about the speedup + graph

- diepere uitleg implementatie met futures
- uitleg schaalbaarheid van systeem, met diagram van uitgebreid systeem
- hypothese over snelheidsoptimalisatie

Content die hendrik wil:

- What did we want to address
- How did we design the actors, where was the caching, where which processing steps
- What was the performance (did we speed up response time?)
- A design or system diagram

5 Visualization in the browser with D3

- What did we want to show?
- How did we implement it (D3, what graph parameters)
- How did we emphasize characteristics (e.g. central projects)
- An example result

6 Conclusions and further research

- Did it work
- Is it responsive?
- Did we show interesting things
- What more interesting things could fit in this framework?

We believe the given approach can be easily adapted to include other parameters or show other relations in the data.

- Optimization by only considering weeks

References

J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

G. Gousios and D. Spinellis. Ghtorrent: Github's data from a firehose. In *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*, pages 12–21. IEEE, 2012.

R. Lämmel. Google's mapreduce programming model – revisited. *Science of Computer Programming*, 70(1):1–30, 2008.

M. Odersky. Pimp my library. *URL http://www.artima.com/weblogs/viewpost.jsp?thread=179766*, 2006.

M. Odersky and A. Moors. Fighting bit rot with types. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 4, pages 427–451, 2009.

R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with sawzall. *Scientific Programming*, 13(4):277, 2005.

Appendix

A Scala collection Aggregator example