GHTorrent: Github's Data from a Firehose

Georgios Gousios and Diomidis Spinellis
Department of Management Science and Technology
Athens University of Economics and Business
Athens, Greece
{*gousiosg,dds*}*@aueb.gr*

***Abstract*—A common requirement of many empirical software engineering studies is the acquisition and curation of data from software repositories. During the last few years, GitHub has emerged as a popular project hosting, mirroring and collaboration platform. GitHub provides an extensive REST API, which enables researchers to retrieve both the commits to the projects' repositories and events generated through user actions on project resources. GHTorrent aims to create a scalable off line mirror of GitHub's event streams and persistent data, and offer it to the research community as a service. In this paper, we present the project's design and initial implementation and demonstrate how the provided datasets can be queried and processed.**

***Keywords*-dataset; repository; GitHub; commits; events**

I. Introduction and Motivation

When conducting empirical research with data from software repositories, a typical step involves the downloading of the project data to be analysed. Obtaining data from open source software (OSS) project repositories is a tedious exercise lacking scientific value, while the obtained datasets are often non-homogeneous which makes further analysis difficult. An indicative example of data disparity is that of developer identities; as projects use several management tools to support development, it can be very difficult to track a single developer's trails across those tools reliably, because the developer can employ multiple login identities. Moreover, while product data (mainly source code) is straightforward to retrieve and process, the same does not hold for process data (issues, wikis etc., because what researchers can usually obtain from repositories are static snapshots containing little evolutionary information.

GitHub is a relatively new project hosting site, which, due to its developer-friendly features, has enjoyed widespread acclaim and adoption. GitHub bases its operation on the *git* [1] revision management system, not only for source code versioning but also to handle the project's wiki. GitHub also facilitates project contributions from non-team members through features such as forking, which creates a private copy of a *git* repository, and pull requests, which organize a series of commits to a forked repository into a mergeable patch for the source repository. Moreover, GitHub offers the usual gamut of software forge facilities, such as a bug tracker, a wiki, and developer communication tools. What makes GitHub particularly attractive for the repository mining community, is that it provides access to its internal data stores through an extensive REST [2] API,[1] which researchers can use to access a rich collection of unified and versioned process and product data.

Although GitHub's API is well designed and documented, researchers wishing to use it in order to analyze its data will face a number of challenges.

- GitHub's data is huge (on the terabyte scale, according to our estimation) and an API rate limit of five thousand requests per hour makes the complete download of the data impossible.
- The overall schema of GitHub's data is not documented. A researcher wishing to explore it would have to reverse engineer it from the available REST requests and the corresponding JSON [3] replies.
- The API does not provide facilities to obtain collections of the data's key entities; the existing API calls mainly allow navigation from one entity to another.
- Events are only provided as streams, which disappear into a sliding window of 300 entries.

In this work, we present GHTorrent a service that gathers event streams and data from the GitHub hosting site and provides those data back to the community in the form of incremental MongoDB data dumps distributed through the peer-to-peer BitTorrent [4] protocol.

The main contributions of this work are the following.

- The documentation of GitHub's schema, the use of the REST API to navigate through its elements, and the relationship between events and static entities (Section II).
- The design and implementation of an extensible infrastructure for the collection of all (the normally fleeting) events exposed through GitHub's API (Sections III and IV). The collected events are browsing roots for collecting virtually all information in a project's timeline.
- The collection of half a year's worth of commits into a permanent store. These can now be downloaded and used as starting points for deep crawling operations. Through these recorded events one can combine the GHTorrent database with the GitHub API to examine

[1] http://developer.github.com/v3/

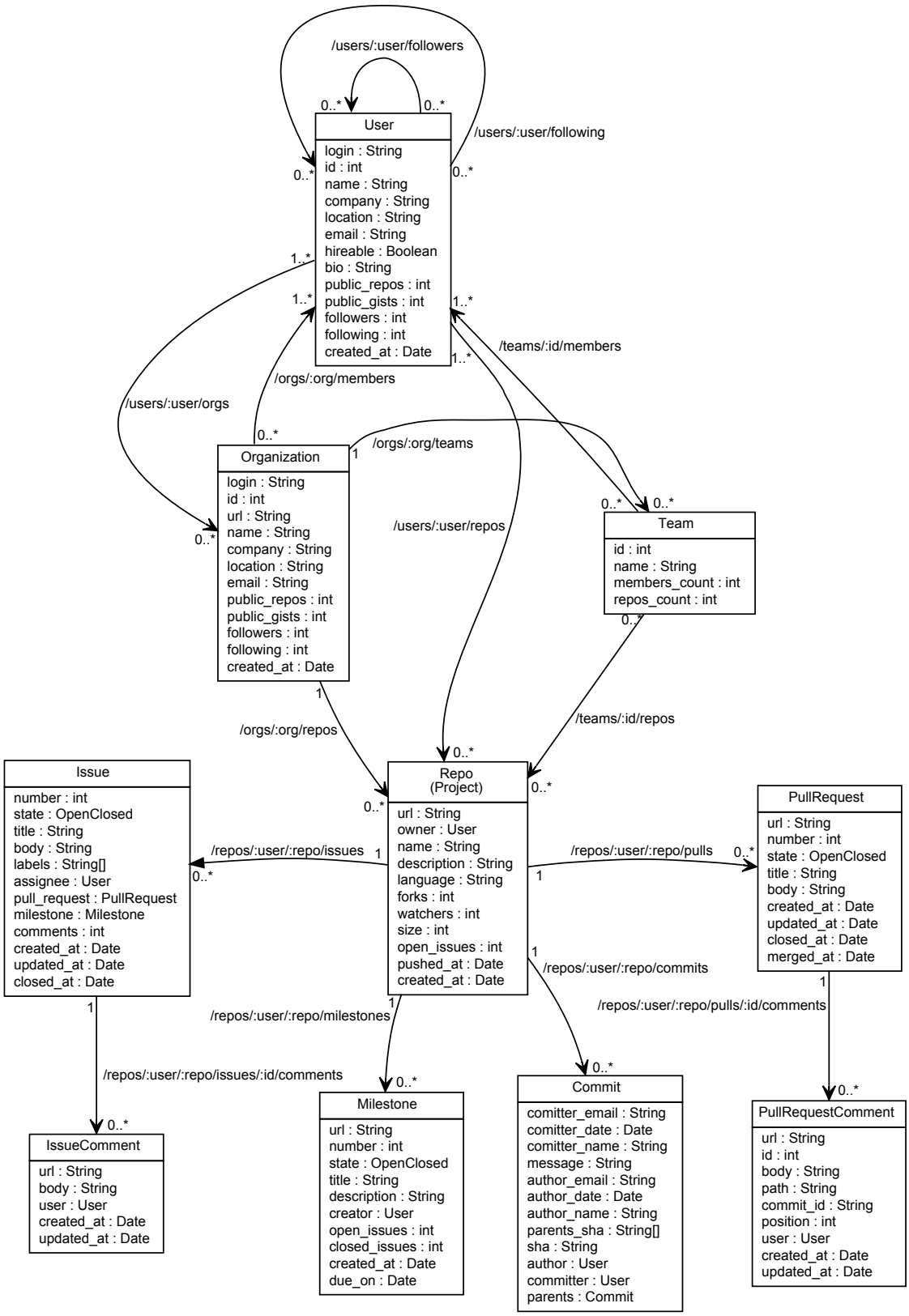


Figure 1. An overview of GitHub's data schema and API

more than 200 thousand users and 400 thousand repositories.
- The realization of a scalable mechanism for providing researchers with GitHub's data, based on distributing incremental data dumps using a peer-to-peer protocol.
- The provision of data that can track developers through both a project's process (issue tracking, wiki) and its corresponding source code without resorting to heuristics. In Section V, we present evidence that the GHTorrent dataset can be used towards answering non-trivial research questions in a scalable manner.

II. GitHub's Static Schema and Event Stream

An overview of the most important elements of GitHub's static data schema appears in Figure 1. A *user* who signs into GitHub and can create a *repository* under her account, or she can manage repositories belonging to other users by participating to an *organization* or a *team* defined within the organization to manage repository access rights. A repository is the equivalent of what other hosting providers call a project. A user can *commit* changes to the repository; these may originate from another author. An *issue* associated with the repository documents a problem or a feature request. It can be associated with a *milestone*, which refers to a collection of issues that must be closed on a target date. A *pull request* associated with the repository presents some changes that a user can integrate into the repository. Both issues and pull requests can have comments associated with them. Finally users can follow each other and see the followed user's activity.

GitHub provides a REST API to navigate between the various entities. Figure 1 lists the navigation paths and the corresponding URLs as edges between the graph's elements. Other links, such as an issue's assignee, a commit's author, or a milestone's creator, are embedded directly within the returned elements and can be identified in the Figure through their corresponding data types.

In Figure 1, a commit's user, author, and parents appear on a grey background to signify that the philosopher's stone of software repository miners, namely the relationship between commits and other repository data, is not always directly available on a commit's fields. Commits appearing as push events (see below) or retrieved through their SHA value are associated with an author and a committer that are only given in free text form. For instance, a commit may contain

```
"committer": {
  "email": "jason@curlymedia.com",
  "date": "2012-01-26T08:19:43-08:00",
  "name": "Jason Tempestini"
},
"author": {
  "email": "ddrake@dreamingmind.com",
  "date": "2012-01-22T22:52:49-08:00",
  "name": "Don Drake"
}
```

On the other hand the GitHub user associated with the commit is identified by attributes such as the following, which do not appear in the commit's data.

```
"actor" : {
  "gravatar_id" :
    "45d832888fd298d145b91b687130db7f",
  "url" :
    "https://api.github.com/users/curlyjason",
  "id" : 1307159,
  "login" : "curlyjason"
}
```

One can match formal GitHub users with their commits by means of heuristics, such as (approximate) matches on the user's name and email address and hope for the best. Fortunately, as we shall see, the dynamic view of GitHub provided through events contains data for making the corresponding connection. In addition, commits retrieved using the repos/:user/:repo/commits URL contain both fields. Even then, the author field is correctly filled-in, only if the author submitted the change through GitHub.

In parallel with the access of GitHub's persistent data the API also provides a mechanism for retrieving a stream of events through a /events REST API request. This stream provides an overview of GitHub's activity, and is normally used to trigger various actions and update status displays. Crucially for our goals, we can also use this event stream as a starting point for mining GitHub's static repository and also for associating commits with specific users and repositories. Figure 2 depicts how the GitHub API events are associated with many of the schema's persistent entities. Most events provide a user and a repository, and some, like *ForkEvent* provide two. Although an organization is part of all events' data in practice few of the events link to an organization. Three events, *Issues*, *IssueComment*, and *PullRequest* link to an issue.

Most importantly, two events, *ForkApply* and *Push* link to a commit. For instance, the following *PushEvent* event identifies both a GitHub user and a repository.

```
"repo" : {
   "name" : "codl/autoplay",
   "url" : "https://api.github.com/repos/codl\
/autoplay",
   "id" : 815830
},
"actor" : {
  "gravatar_id" : "732afa12eb18eba9b8888b0d69\
ba2f7d",
  "url" : "https://api.github.com/users/codl",
  "id" : 315139,
  "login" : "codl"
},
"payload" : {
  "commits" : [
    {
      "sha" : "6f80371734f3707e1e95638d7f0ef1\
2b54c6ea8c",
```

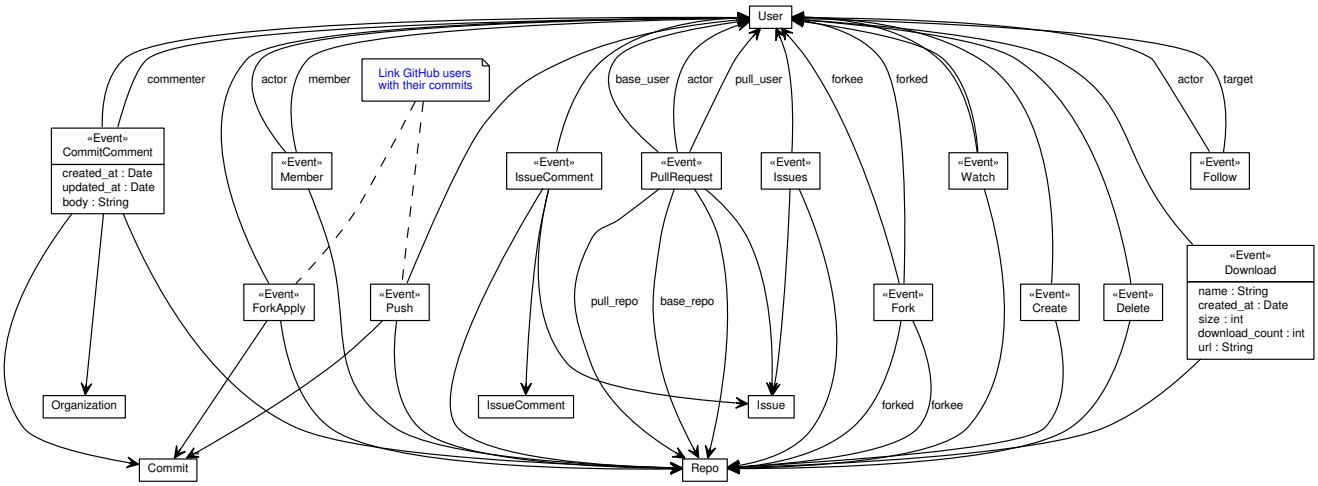


Figure 2. Obtaining handles to persistent elements through the Event API

Through the payload's SHA values (a cryptographic hash uniquely identifying each commit) one can also issue a REST API GET request on the resource https://api.github.com/repos/codl/autoplay/git/commits/6f80371734f3707e1e95638d7f0ef12b54c6ea8c to obtain the associated commit.

```
{
  "committer": {
    "email": "codl@aquageek.net",
    "date": "2012-01-26T06:45:10-08:00",
    "name": "Corentin Delcourt"
  },
  "message": "fix #7 : \"dies if mpd is stop\
ped\"\n",
  "author": {
    "email": "codl@aquageek.net",
    "date": "2012-01-26T06:40:56-08:00",
    "name": "Corentin Delcourt"
  },
  "parents": [
    {
      "sha": "7758a468924a28487c0a35409e9a0e\
edebf83f63"
    }
  ],
  "sha": "6f80371734f3707e1e95638d7f0ef12b54\
c6ea8c",
  "tree": {
    "sha": "1dd47cae45408abf26551e5073d3d6a2\
dc01533d"
  }
}
```

III. The Design of GHTorrent

According to its current front page, GitHub is hosting almost 4 million repositories from 1.5 million users. Even though the majority of the repositories are inactive forks of well known projects (e.g. the `node.js` project has 13,000 forks), GitHub receives more than 70,000 commits per day (see Figure 4(a)). In comparison, SourceForge retrieves an average of 6,000 commits per day.[2] As a result, the processing, storage and network capacity requirements to process the entire GitHub commit stream can be quite steep. In addition, the event stream imposes temporal restrictions; if the mirroring is too slow and events are lost, then the static data view at any point cannot be reconstructed through event replication, as initial data pointers will be lost. Furthermore, in order to be useful, the format of mirrored data distribution should enable the community to incrementally update their copies when new versions of the data become available while not taxing the project's host bandwidth or requiring to perform any queries to GitHub.

GHTorrent is designed to cope with scale through the distribution of the data collection and its replication among research teams. GHTorrent is designed to use multiple hosts for collecting the GitHub static data and dynamic events, thus overcoming the rate limit restrictions. In the future a network of co-operating research teams, where each of which uses its own infrastructure to mirror diverse parts of the GitHub data, can cooperate to increase the data's coverage. The collected data are then distributed among the research teams in the form of incremental data dumps through a peer-to-peer protocol. The data's replication among those downloading it, ensures the distribution mechanism's scalability. Three types of dumps are necessary for the reconstruction of a full project's view.

- *Events*, which contain aggregated pointers to the data generated through actions performed by users on the repositories. They are useful for tracking the project's timeline.

[2]As can be retrieved through SourceForge's home page header at http://sf.net/

- *Raw data*, which contain the data pointed to by the events. These are divided among the various entities stored in GitHub: users, organizations, teams, commits, issues, repositories, pull requests, issue comments, milestones, and pull request comments.
- *State data*, which are generated by post-processing the raw data to recreate the project's state at certain points in the project's timeline.

The GHTorrent project was designed to evolve through the following phases.

1) We collected commits through GitHub's RSS feed (initially) and its commit stream page when the RSS feed became unavailable. This effort allowed us to fine-tune the collection mechanism, the database, and the project's data distribution protocol. The collected data form the basis for the 2011 datasets.
2) We changed the collection to be based on events. We collect data for commits and watch events and plan to add all other types of events in the collection queue. This is an ongoing effort and forms the basis for the datasets distributed from 2012 onward.
3) We expanded the collection operation to use event data as the starting point for deep crawling operations to collect raw data. This effort started in February 2012 and its results will appear in forthcoming datasets.
4) We plan to engage with the research community to overcome growth challenges by splitting the collection and processing work among multiple sites on project partner's premises. The work will include additional crawling and the processing of raw data to create state data representations. The project's peer-to-peer distribution mechanism is ideal for engaging multiple teams with minimal central coordination effort.

Data is distributed among researchers using the BitTorrent [4] peer-to-peer protocol as incremental database compressed dump files named after the month they appeared. Each file contains one or more sets of data; nodes of the database's hierarchy. This allows the future splitting of the data collection and processing among more teams. A time stamp stored with each entry ensures that there is an exact split of the data between the incremental dumps. Through the successive loading of all the incremental GitHub dumps a researcher can obtain a complete copy of the project's data.

In the following sections, we describe how we implemented the software to provide data for the project's first three phases.

IV. Implementation

The mirroring scripts were designed from the ground up to work in a cloud computing environment, with multiple crawlers retrieving data from the GitHub API in parallel, while respecting the per-IP resource usage limits. All mirroring operations are performed twice by instances of each

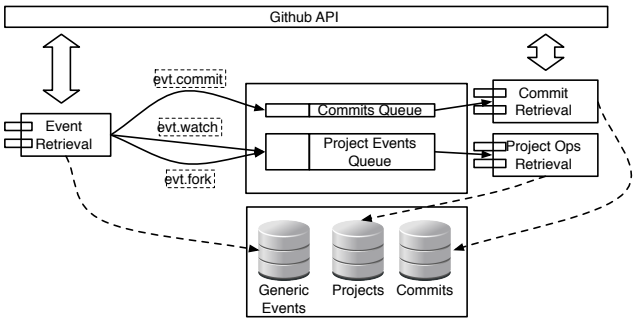


Figure 3. Data retrieval system architecture

specific crawler running in two separate data centers, to ensure that all data generated by GitHub is accurately persisted in the system's data store. Retrieval of data is performed in stages, which are connected through the pipeline paradigm over a message queue. The retrieved data are stored in a document-oriented NoSQL database, to accommodate future changes to the data returned by GitHub. An overview of the mirroring architecture is presented in Figure 5.

The data retrieval is performed in two phases: in the event retrieval phase, a crawler queries the GitHub events API, and stores new events in the events data store. It then sends the payload of each new event as a topic exchange message to the system's message queue, along with an appropriate routing key. In AMQP-based queueing systems, such as RabbitMQ, which we are currently using, topic exchanges enable clients to declare queues selecting the required messages by applying regular expressions on message routing keys. Therefore, during the event processing phase, clients setup queues and select interesting messages for further processing; for example, the commit retrieval client selects all messages routed as commits and then queries GitHub for the commit contents. Similarly, the project details retrieval agent can setup a queue that collects all watch and fork events to query GitHub and update the project's state in the local database mirror.

Both processing phases can run independently on multiple hosts, to ensure fault tolerance in case of a host or network problems. Clients in the data processing phase can run either in load balancing mode, thus re-using existing queue declarations, or in high availability mode by declaring client-exclusive queues. Processing is not necessarily performed online; as raw events are stored in the database, a researcher can reprocess all events by requeueing the stored events as necessary. This allows for new types of event processing clients to be developed at any time, as long as the GitHub data formats remain unchanged.

A problem that we had to deal with was adhering to GitHub's API usage restrictions, which currently limits hosts to 5.000 requests per hour. The problem mostly manifests

during the second phase of data retrieval operations, since for each retrieved event that is queued, at least one and usually more additional requests to the GitHub API must be performed. As our top priority was to ensure that all events are retrieved, we run the event retrieval component in a standalone instance; the rate of requests is dynamically moderated by comparing for each request the number of new events to the number of events already retrieved. We found that a rate of 450 requests per hour is adequate to keep up with GitHub's event stream. For the second phase data of retrieval operations, we employ a strict rate limit of 80 requests per minute to comply with GitHub's policy. This rate is usually not adequate to retrieve all data from a single host and messages fill the queue, unless more hosts are added in load balancing mode.

The data is stored in a MongoDB database [5]. MongoDB is known to be able to handle large collections of unstructured data. It can scale to multiple hosts by evenly distributing both data storage and data querying operations across multiple hosts. Moreover, MongoDB offers some support for the Map-Reduce [6] data processing model, which has been received with interest in the repository mining community [7]. Given the amount of data available, the strict consistency guarantees offered by traditional databases can be relaxed in order to scale data processing on multiple hosts [8]. The data is stored in MongoDB's native BSON format, which we then exploit to generate incremental data dumps efficiently.

To distribute the mirrored data, we opted for periodic archiving of the data and delivery through the P2P BitTorrent protocol or direct downloads through a web server. The choice of BitTorrent as a medium for distribution was directed by the fact that it permits the distribution of bandwidth load across participating partners. Also the protocol is more widely used than niche content distribution technologies such as Tahoe [3] or MogileFS[4] which in turn incurs less administrative effort to end users. Moreover, sophisticated BitTorrent clients or custom made scripts can use RSS feeds to discover new torrent files on our site, automatically download the linked data dumps and import them to MongoDB, enabling full automation of the data distribution process.

To generate the data archives, we query MongoDB for data in time windows, currently one month long. Every archive contains a dump of the data of each MongoDB collection within the specified time window. For each archive, we construct a torrent file which contains a list of tracker servers and content hashes for file chunks.[5] The torrent files are then linked in a directory monitored by the *rtorrent* program, which picks them up and registers them to the specified trackers. The torrents are then ready for download by the community.

[3]https://tahoe-lafs.org/~warner/pycon-tahoe.html

[4]http://danga.com/mogilefs/

[5]Trackers are central nodes in the BitTorrent network which regular nodes query in order to discover peers that offer chunks of the downloaded file.

Table I
VARIOUS METRICS IN THE CURRENT DATASET

Metric	Value
Number of commits (since Aug 2011)	8,817,685
Number of events (since 1 Feb 2012)	4,512,000
Number of active repositories	424,995
Number of active committers	203,470
Average commits per day	41,300
Average events per day	94,189
Size of commits for typical (.bz2)	80MB
Size of events per day (.bz2)	23MB
Size of available uncompressed data	76GB
Size of available compressed data	18GB

V. USING THE SERVICE

To use the service as a peer, a researcher must first download a data snapshot and import it in a locally installed version of MongoDB. The available data collections are currently commits and events (their sizes are indicated in Table I), but as the project enters into phase 2 and data from events are beginning to get processed, more collections will follow. For each collection, the project distributes a series of monthly dumps. Processing the provided data is relatively easy with any mainstream programming language. From our experience, untyped languages that provide inherent JSON support, such as Python and Ruby, are very well suited for prototype building.

A. Interesting Facts

Table I presents various size metrics about the current status of the GHTorrent dataset. In about eight months of operation, our crawler collected roughly nine million individual commits in 425 thousand repositories. Figure 4(a) shows the number of commits collected per day. The plot is divided in three periods; up to end Nov 2011, we polled GitHub's timeline RSS feed in order to retrieve the commits. After GitHub deprecated the RSS feed, we scraped the generated timeline page, which was not refreshed as often. After version three of the Github API was introduced, we switched to collecting commits through *PushEvents*. This lead to the collection of more than 70 thousand commits per day. The short period of apparent inactivity at the end of Feb 2012 was the result of a bug in the event mirroring script, which manifested in both event retrieval nodes. We have since employed mechanisms to monitor the daily event retrieval activity in order to make sure that such problems will not remain undetected for more than a few hours.

Table I also shows that the number of generated events is staggering; every day more than 90 thousand events, the majority of which, as Figure 4(b) shows, are *PushEvents* are added to our database. Even if each event contained only one data item worth retrieving with an additional request, it

would take on average 18 hours to retrieve all pointed items, given GitHub's API request rate limit for a single host. In practice most events, especially *PushEvents*, contain more than one events worth retrieving, which means that a single host cannot keep up with GitHub's event stream.

GitHub is collaborative platform used by developers around the world. As this paper's second author has shown in previous work [9], work never stops in global software development projects. To investigate in what extend this happens on the GitHub dataset, we plotted the distribution of commits within the hours of the day for all commits in the project's archives. The times for the commits where adjusted to UTC. What we see in Figure 4(c) is that development does indeed take place around the clock, with a peak around the time that both European and American developers are active. This finding is in agreement with Takhteyev and Hilts [10], who, using geocoding information based on developers' email domains, found that the majority of developers indeed reside in the US (43.1%) or Europe (20,3%). Interestingly, our data also show a feature normally associated with paid-work employment habits: an energetic start on a Monday with productivity declining as the week progresses (Figure 4(d)). (However, we suspect that work during weekends would be lower in projects not staffed by volunteer developers.)

B. Estimating Average Commit Sizes

One of the usually discussed differences among programming languages is the brevity they afford. It is widely believed [11], [12] that functional programming languages allow the expression of more concise code, while languages such as Java and C++ are notorious in programming circles for being more verbose. An approximation of the verbosity of a language might be given by the commit sizes that programmers usually produce. While the hypothesis is fragile as the size of a commit might depend on many factors, such as project guidelines, programming style, framework verbosity, indications might be obtained if a large number of data points (commit sizes) is explored. The following examination only serves as an example of how the GHTorrent dataset can be used to answer an interesting question.

To study the commit size across programming languages, we used a snapshot of the dataset from late July 2011 up to mid-March 2012. The dataset consisted of about 8,500,000 commits. To experiment with large scale computation, we implemented a custom Map-Reduce process. In the map phase, a script selected the *git* commit SHA-1 tokens and appended them to a work queue; during the reduce phase, another script retrieved the commit contents from the dataset, identified the language that most files where written in (through their filename's extension) and calculated the length of changed (both added and deleted) lines per file, which was then saved to a text file. To run the experiment, we used a combination of two physical machines in our cluster and three virtual machines in a remote data center, all of which were granted access to both MongoDB and to the work queue running on the same physical machine. Throughout the experiment neither MongoDB, nor RabbitMQ consumed more than 10% of CPU power each.

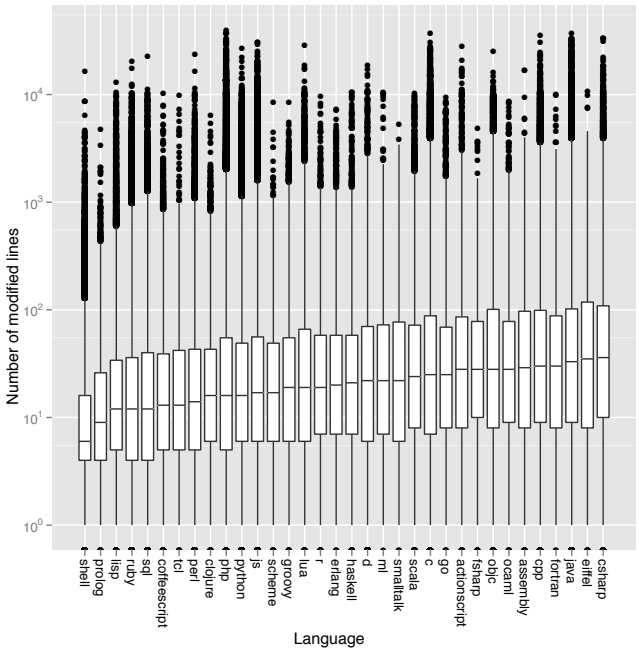
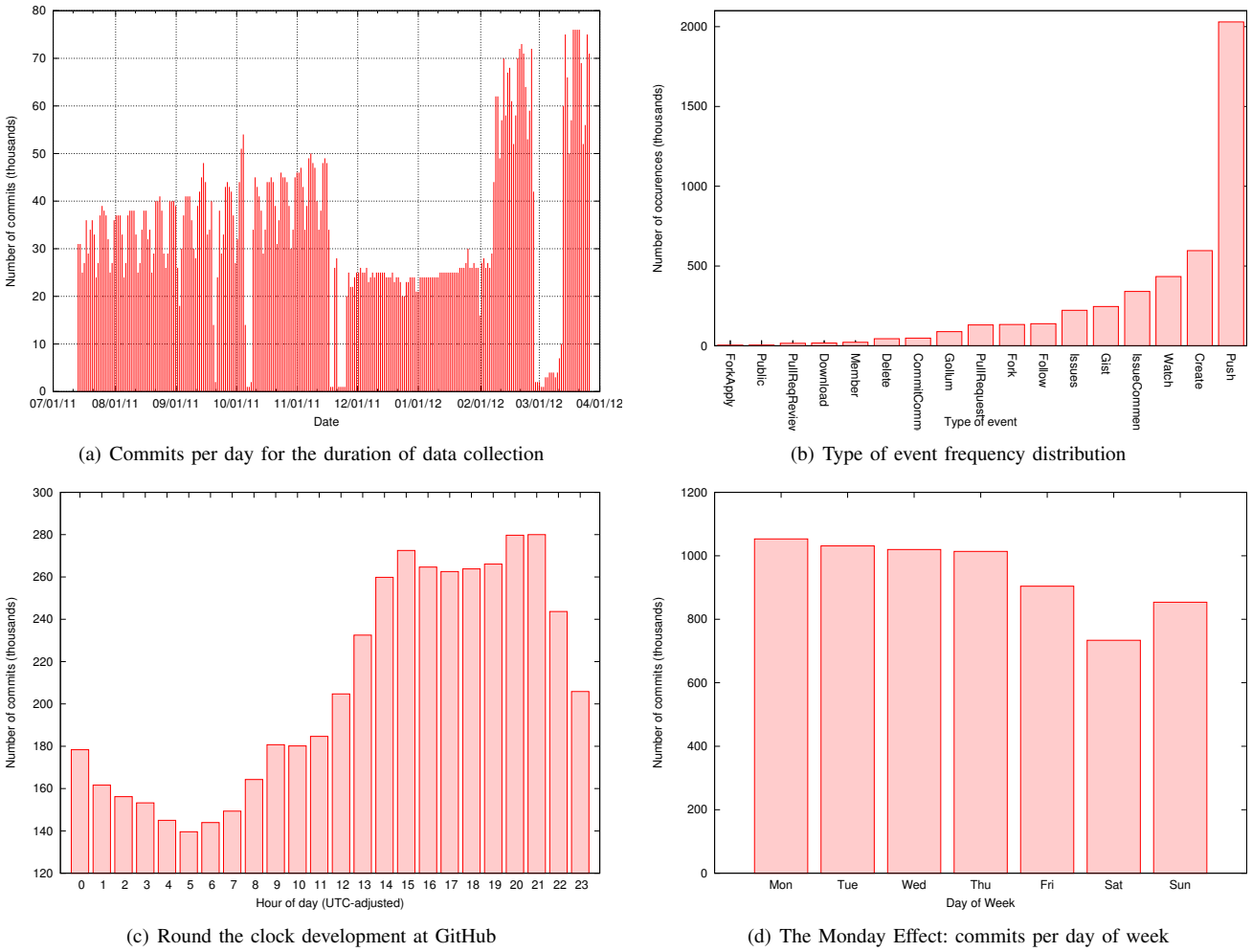


Figure 5. Commit sizes (quartiles and outliers) across languages

The results of our case study can be seen in Figure 5. For each language, we plotted a quartile boxplot along with its outliers. The box plots were then sorted by median value and the dataset outliers were added. While no conclusive proof can be extracted from this chart, we can observe a tendency of languages featuring functional programming characteristics, such as Ruby, Lisp and Javascript to exhibit a lower median and range values than languages that are allegedly verbose, such as Java or C++. However, the real value of this experiment is that it exhibited the feasibility, usefulness, and scalability of our approach. Using a ready made data set, very simple Ruby scripts and powerful processing abstractions (work queues, Map-Reduce), we were able to distribute the processing load on a cluster of machines and obtain a result in minutes.

VI. Related Work

This work is not the first to gather data from software repositories and offer them back to the community. Howison et al [13], in the FLOSSmole project, collected metadata from OSS projects hosted on the Sourceforge site and offers both the dataset and an online query interface. The dataset consists of metadata about the software packages, such as numbers of downloads or the employed programming



(a) Commits per day for the duration of data collection

(b) Type of event frequency distribution

(c) Round the clock development at GitHub

(d) The Monday Effect: commits per day of week

Figure 4. Facts extracted from the current version of the GHTorrent dataset

language. Since its original release, the FLOSSmole dataset has been extended to include data from various OSS repositories, albeit not in a unified format. The Flossmetrics project [14] provides a dataset consisting of source code size, structure and evolution metrics from several OSS projects. The Sourcerer project [15], apart from the analysis tools, also provides a pre-processed dataset of code structure metrics from thousands of Java projects. Similarly, the Alitheia Core project [16], provides an analysis platform and an accompanying dataset containing process and product metrics. From the datasets mentioned above, FLOSSmole has seen the widest adoption outside the research team that produced it [17].

The bulk of the work on the repository mining community is being carried out with data from centralized repositories. As OSS projects move en masse to decentralized source code management repositories, researchers are slowly beginning to explore the new opportunities provided by data collected from such systems. In reference [18], Bird et al. provided the first complementary account of the difficulties and opportunities of mining Git repositories. Rahman and Devanbu exploited [19] Git's strong authorship capabilities to study the impact of ownership and developer experience on software defects. Capilupi and Cortázar used [20] Git's ability to track commit time at the developer's cite to produce a round the clock development bar chart similar to ours (Figure 4(c)). Finally, Barr et al. [21] explored the use of branches, a commonly cited advantage of distributed version control systems, as a factor of distributed version control adoption.

A common task when doing research with repositories is that of tracking developer identities across data sources. The problem manifested itself in the early repository mining days, as the software configuration management systems relied on user names to identify developers, while mailing lists and bug tracking software used the developers' emails. The problem has been originally described by Robles and Barahona in reference [22]. The authors describe simple

heuristics to match developer identities across a project's Subversion repository and its mailing lists. Bird et al. [23] used similarity metrics to semi-automatically align multiple email identities to unique users. In reference [16], we expand on this work by adding approximate string matching heuristics, which slightly improved the identity matching score. A more thorough treatment is described in reference [24]; the author uses a set of string and partial matching heuristics similar to ours, but allows them to be weighted in order to compensate for different developer user name generation conventions across projects. The data in the GHTorrent dataset offer the potential for more precise analysis across repositories, as *git* uses emails to identify developers, while both bug descriptions and wiki entries coming from the same developers feature their identity details. Most importantly, and to the best of our knowledge uniquely among software repository mining efforts, as we showed in Section II, the datasets we provide enable researchers to map commits to GitHub users, which in turn enables interesting correlation possibilities between commits and issues.

VII. Conclusions and Future Work

In this paper, we described the GHTorrent project, an effort to bring GitHub's rich product and process data to the hands of the research community, in a scalable manner. The project has already amassed several GB worth of data and is distributing those over BitTorrent. The Github dataset has a strong potential for providing interesting insights in areas including but not limited to community dynamics, global software engineering, distributed collaboration and code authorship and attribution.

The GHTorrent project is in the initial stages of development. The crawlers that GHTorrent uses currently retrieve raw event contents and store them in MongoDB collections according to the type of retrieved data. To reconstruct Github's data schema as presented in Figure 1, we need to replay events on top of an initial state, which might not be available in all cases. We are currently investigating ways to both efficiently apply events in order to recreate the static view and to go back in a user's or repository's history in order to retrieve its initial state. We also plan to automate the generation and distribution of torrent files through RSS feeds and scripts that will monitor those and automatically download and update remote databases. Finally, an interesting challenge is to write tools or platforms that are able to process the vast amount of data that GHTorrent offers.

The project is intended as a community effort. We are actively seeking contributions that will enhance the collected data's utility to the research community. Ideas might include further processing and extraction of new metadata (e.g. project categorization by language) or visualizations of interesting data properties (e.g. growth rate, events rate). The full source code and instructions on how to download and use the mirrored data can be obtained at https://github.com/gousiosg/github-mirror.

Acknowledgements

The authors would like to thank Panagiotis Louridas for his help in creating the commit size distribution plot, Vassileios Karakoidas for providing useful advice and the anonymous reviewers for their targeted comments on earlier drafts of this paper.

This work is supported by the European Regional Development Fund (ERDF) and national funds and is a part of the Operational Programme "Competitiveness & Entrepreneurship" (OPSE II), Measure COOPERATION (Action I).

References

[1] J. Loeliger, *Version control with Git: Powerful Tools and Techniques for Collaborative Software Development.* Sebastopol, CA: O'Reilly Media, Jun. 2009.

[2] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, 2000.

[3] D. Crockford, "RFC 4627 — the application/json media type for JavaScript object notation (JSON)," IETF RFC. Available online http://tools.ietf.org/html/rfc4627, Tech. Rep.

[4] B. Cohen, "The BitTorrent protocol specification," Available online http://www.bittorrent.org/beps/bep_0003.html, BitTorrent.org, Jan. 2008, current 6 February 2012.

[5] K. Chodorow and M. Dirolf, *MongoDB: The Definitive Guide*. Sebastopol, CA: O'Reilly and Associates, 2010.

[6] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[7] W. Shang, B. Adams, and A. E. Hassan, "Using Pig as a data preparation language for large-scale mining software repositories studies: An experience report," *Journal of Systems and Software*, vol. In Press, Corrected Proof, 2011.

[8] P. Helland, "If you have too much data, then 'good enough' is good enough," *Communications of the ACM*, vol. 54, pp. 40–47, Jun. 2011.

[9] D. Spinelis, "Global software development in the FreeBSD project," in *International Workshop on Global Software Development for the Practitioner*, P. Kruchten, Y. Hsieh, E. MacGregor, D. Moitra, and W. Strigel, Eds. ACM Press, May 2006, pp. 73–79.

[10] Y. Takhteyev and A. Hilts, "Investigating the geography of open source software through GitHub," Available online http://takhteyev.org/papers/Takhteyev-Hilts-2010.pdf, University of Toronto, Aug. 2010, current 10 February 2012.

[11] P. Hudak and M. P. Jones, "Haskell vs. Ada vs. C++ vs. AWK vs. ...an experiment in software prototyping productivity," Yale University, Dept. of CS, New Haven, CT, Tech. Rep., Jul 1994.

[12] C. Ryder and S. Thompson, *Trends in Functional Programming*. Kluwer Academic Publishers, September 2005, ch. Software Metrics: Measuring Haskell.

[13] J. Howison, M. Conklin, and K. Crowston, "Flossmole: A collaborative repository for FLOSS research data and analyses," *International Journal of Information Technology and Web Engineering*, vol. 1, no. 3, pp. 17–26, 2006.

[14] I. Herraiz, D. Izquierdo-Cortazar, F. Rivas-Hernandez, J. González-Barahona, G. Robles, S. Dueñas Dominguez, C. Garcia-Campos, J. Gato, and L. Tovar, "Flossmetrics: Free/libre/open source software metrics," in *CSMR '09: 13th European Conference on Software Maintenance and Reengineering*, march 2009, pp. 281–284.

[15] J. Ossher, S. Bajracharya, E. Linstead, P. Baldi, and C. Lopes, "SourcererDB: An aggregated repository of statically analyzed and cross-linked open source Java projects," in *Proceedings of the International Workshop on Mining Software Repositories*. Vancouver, Canada: IEEE Computer Society, 2009, pp. 183–186.

[16] G. Gousios and D. Spinellis, "A platform for software engineering research," in *MSR '09: Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories*, M. W. Godfrey and J. Whitehead, Eds. IEEE, May 2009, pp. 31–40.

[17] F. Mulazzani, B. Rossi, B. Russo, and M. Steff, "Building knowledge in open source software research in six years of conferences," in *Proceedings of the 7th International Conference on Open Source Systems*, S. Hissam, B. Russo, M. de Mendonça Neto, and F. Kon, Eds., IFIP. Salvador, Brazil: Springer, Oct 2011, pp. 123–141.

[18] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu, "The promises and perils of mining Git," in *MSR '09: Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories*, M. W. Godfrey and J. Whitehead, Eds., Vancouver, Canada, 2009, pp. 1–10.

[19] F. Rahman and P. Devanbu, "Ownership, experience and defects: a fine-grained study of authorship," in *ICSE '11: Proceedings of the 33rd International Conference on Software Engineering*. New York, NY, USA: ACM, 2011, pp. 491–500.

[20] A. Capiluppi and D. Izquierdo-Cortázar, "Effort estimation of FLOSS projects: a study of the Linux kernel," *Empirical Software Engineering*, pp. 1–29, 10.1007/s10664-011-9191-7.

[21] E. T. Barr, C. Bird, P. C. Rigby, A. Hindle, D. M. German, and P. Devanbu, "Cohesive and isolated development with branches," in *Proceedings of the International Conference on Fundamental Approaches to Software Engineering*. Springer, 2012.

[22] G. Robles and J. M. González-Barahona, "Developer identification methods for integrated data from various sources," in *MSR '05: Proceedings of the 2005 International Workshop on Mining Software Repositories*. New York, NY, USA: ACM, 2005, pp. 1–5.

[23] C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan, "Mining email social networks," in *MSR '06: Proceedings of the 2006 International Workshop on Mining Software Repositories*. New York, NY, USA: ACM, 2006, pp. 137–143.

[24] W. Poncin, "Process mining software repositories," Master's thesis, Eindhoven University of Technology, Aug 2010.